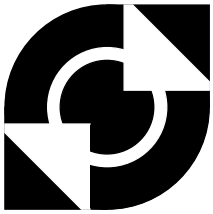
Tele-Informatics
&
Open Systems

SMILE user manual

release 4.0

Henk Eertink



University of Twente
Department of Computer Science
& Department of Electrical Engineering
Tele-Informatics and Open Systems Group
P.O.Box 217, 7500 AE Enschede, The Netherlands

Contents

1 Introduction **1**

2 A quick tour through SMILE **3**
2.1 Specification and simulation tree 3
2.2 ADT Interface 6
2.3 Info window 8
2.4 Command interface 10

3 User Manual **13**
3.1 Simulation trees and EFSMs 14
3.2 Using Tcl to build a simulation tree 19
3.3 Executing test cases 21
3.4 Instantiating variables 22
3.5 Ignoring behaviour expressions 23
3.6 Execution towards a behaviour expression 23
3.7 Proving that particular actions exist 24
3.7.1 Defining actions 24
3.7.2 Using actions to execute a specification 25
3.7.3 How does this work 27
3.8 Narrowing 28

4 Reference manual **31**
4.1 SMILE Specification window 31
4.1.1 Simulate pull-down menu 31
4.1.2 Options pull-down menu 33
4.1.3 Misc pull-down menu 34
4.1.4 Help pull-down menu 35
4.2 SMILE Simulation Tree Window 35
4.2.1 Execute pull-down menu 35
4.2.2 Execute options pull-down menu 38
4.2.3 Event pull-down menu 40
4.2.4 Node pull-down menu 42
4.2.5 Display pull-down menu 43
4.2.6 Misc pull-down menu 43
4.2.7 Traversing the simulation tree 45
4.3 SMILE Adt Interface window 46

4.3.1 ADT specification window 46
4.3.2 Goal window 47
4.3.3 Solution window 49
4.4 Info Window 50
4.5 Command Window 50

5 Programming SMILE **51**
5.1 Variables with special meaning 51
5.2 Commands 53
5.3 Predicates 57
5.4 Some example Tcl-programs 58

6 Customizing SMILE **59**
6.1 Customizing the display 59
6.2 Memory management 60
6.3 Narrowing constants 60
6.4 Setting constants for everybody 61
6.5 Out of Memory 61
6.6 Adding commands to SMILE 62

Chapter 1

Introduction

SMILE stands for *Sy*M*bolic* *I*nteractive *L*OTOS *E*xecution. It is a tool that allows one to execute a LOTOS specification, and to analyse the state/event space denoted by that specification. It offers a number of functionalities:

- Execution of arbitrary parts of the specification
- Single stepping
- Automatic symbolic execution
- Analysis of the abstract data type part of a specification
- Reports on executability of the abstract data type part
- Translation of behaviour expressions into finite state machines
- Execution towards a user-defined state
- Ignoring certain components of the specification while executing
- Verification of certain properties
- Programmable evaluation strategies and analysis techniques

The tool is interactive, and needs an X11 display to run. It is usually invoked from the LOTOS tool environment *lite* by means of the 'Simulate' command.

This manual applies to SMILE version 4.0. It is organized as follows: first, some terminology is introduced by describing a simple SMILE-session. Then the algorithms used in SMILE are explained in more detail in chapter 3. This allows the user to understand why certain operations take more time than others, and how (s)he may be able to change the specification such that execution is more efficient. Chapter 4 can be used as a reference manual, all commands are explained there. In chapter 5 we will describe how SMILE can be programmed, and how new commands can be added to the tool. Finally, in chapter 6 it is shown how the tool can be customized for particular specifications or displays.

Throughout this manual, the following example specification will be used:

```
specification test[g] : noexit
library naturalnumber, boolean
endlib

type my_nat is naturalnumber
opns
  _-_                          : nat, nat                    -> nat
  1, 2, 3                      :                             -> nat
eqns ofsort nat
  forall  x, y : nat
  x - 0 = x;
  0 - x = 0;
  succ(x) - succ(y) = x - y;
  1 = succ(0);
  2 = succ(1);
  3 = succ(2);
endtype (* my_nat *)

behaviour
count_to_3 [g] (0)
where
  process count_to_3[g](x : nat) : noexit:=
       [x lt 3] -> g!x ; count_to_3 [g] (x + 1)
   []  [x gt 3] -> g!x ; count_to_3 [g] (x - 1)
   []  [x eq 3] -> g!x ; count_to_3 [g] (x)
  endproc (* count_to_3 *)
endspec (* test *)
```

This specification contains one process, which counts down (or up, depending on the value of the parameter) to 3, and, when it reaches 3, continues to output 3. It is clear that the transition system of this process consists of only one state (parameterised with the free variable x) and three transitions. This specification is available in the SMILE distribution, as file `demo-smile.lot` and `demo-smile.cr`. The `.lot`-file is the (human-readable) LOTOS file (displayed above), whereas the `.cr`-file is the pre-processed file that is generated by the static semantics checker, and read by SMILE.

In this manual `teletype` font is used for user input and tool output, and **bold** font is used for the names of commands (cf. the titles on the buttons or in the pulldown menus of the user interface).

Chapter 2

A quick tour through SMILE

2.1 Specification and simulation tree

After starting SMILE, a new window is created. This window is depicted in figure 2.1.

The window has two main text areas, a number of pull-down menus containing various commands (the light-grey buttons), as well as two editable fields in which options can be set.

The various areas display the following information:

1. The *error-line* is used to display all error messages, warnings and diagnostics. This is the line directly below the title (in the example it contains the message "`Unfold: ready after 1 steps`").

2. The *specification window* contains the behaviour part of the specification. Initially, only the process-headings are shown (where the top-level behaviour expression receives the name of the specification). In the example, there was only one process defined, viz. `count_to_3`, and the specification was called `test`. The name of the current specification *file* is printed in the title of this sub-window. In the example, the specification file was `demo_smile.lot`.

   In the example, not only the headers of the different processes are shown, but also the behaviour itself. This has been done by setting the **Show behaviour** option in the **Options** pull-down menu.

   The simulation of a certain behaviour expression (or process) can be started by selecting the desired expression with the left mouse-button. The simulation is then started by selecting the **Start** command in the **Simulate**-menu, or by pressing the `Enter`-key. In the example in figure 2.1 a choice expression has been selected for simulation (it is highlighted).

3. The *simulation tree window* contains the events that have been executed in the simulated behaviour expression. Initially, this window is empty. After the **Start**-command has been used to simulate a certain behaviour expression, the simulation tree contains the string `<root-node>`, which corresponds to the root node of the simulation tree. A

Figure 2.1: Main SMILE window

node in the simulation tree corresponds to a state in the specification. The root-node corresponds to the behaviour expression that is currently being simulated. In each state, usually some events are possible. These events are produced by *unfolding* each individual node. A node is selected by clicking on the event that leads to it, and is unfolded by selecting the **Unfold** command in the **Execute** pulldown menu, or by pressing `Enter` on a selected event.

In figure 2.1 the root node has already been unfolded. This resulted in two events, of which the second has been unfolded as well. All events have one predicate attached to them. These predicates are displayed just below the event descriptions. The first event is a value offer at gate `g` with value `succ(succ(succ(succ(Smile_11_0))))`. As the value of `Smile_11_0` is not constrained with any predicate, this denotes an interaction with any value larger than, or equal to, 4. The predicate denotes that the free variable `x_0` is now bound to that particular term. This event corresponds to the LOTOS expression `[ x gt 3] → g!x` in the originally selected behaviour expression: SMILE has computed that the condition `[x gt 3]` can only hold if `x_0` is at least 4. `x_0` is the variable that is generated for the value identifier `x`. Every value declaration in a behaviour expression will result in the generation of a new variable-name; SMILE is symbolic, and continues its computation with the variable instead of the concrete value. The variables that are named `Smile_x_y` are generated by the tool; they denote arbitrary values of the appropriate type.

The simulation (sub)tree associated with the resulting node of an event is displayed below that particular event (with additional indentation). This can be seen at the second event.

The tool sometimes adds an annotation to an event. Three different types of annotations exist:

- `(* Recursion detected *)` which means that the resulting state after this transition has already been visited on the path from the root node to the current event (loop detected).
- `(* Analyzed elsewhere *)` which means that the resulting state after this transition also occurs in another part of the simulation tree and will be expanded there. This occurs for instance when a process consisting of two or more interleaved behaviour expressions is being simulated, say `B=a;B1|||b;B2`. It is then recognised that the resulting state after having executed `a;b` is identical to the state which is reached after having executed `b;a`.
- `(* Instantiated event *)` which means that this event is produced from another event using the **Instantiate** function.

In the example, the second event has the indication `(* Recursion detected *)`. Here SMILE has recognised that no new behaviour will ever occur when the resulting state after this event is also analyzed. Events produced from the resulting state of the first event, on the other hand, will not have this annotation. If this node would be unfolded, the resulting tree would look like

```
<root-node>
    g !succ(succ(succ(succ(Smile_11_0))))
      x_0 = succ(succ(succ(succ(Smile_11_0))))
```

```
        g !succ(succ(succ(succ(Smile_13_0))))
          Smile_11_0 = succ(Smile_13_0)
            g !succ(succ(succ(succ(Smile_14_0))))
              Smile_13_0 = succ(Smile_14_0)
            g !succ(succ(succ(0)))
              Smile_13_0 = 0
        g !succ(succ(succ(0)))
          Smile_11_0 = 0
            g !succ(succ(succ(0)))  (* Recursion detected *)
    g !succ(succ(succ(0)))
      x_0 = succ(succ(succ(0)))
        g !succ(succ(succ(0)))  (* Recursion detected *)
```

As you may infer from this example, SMILE is not able to recognize that the recursive call to `count_to_3` if `x gt 3` does not exhibit any new behaviour. This is because SMILE cannot prove that changing the value of a parameter of a state does not influence the behaviour in that particular state. This subject is discussed in more detail in section 3.1.

But also note that the action that corresponds to the behaviour expression that predicate `x lt 3` is *not* present in the expansion, as SMILE *is* able to prove that the conjunction of `x lt 3` and `x gt 3` is always false. The action denotations from which an event is computed can be selected with the **Find action prefix** command. This command is in the **Event** pull-down menu, and is also bound to the `F4`-key.

The simulation tree can be saved (as well as the EFSM; cf. section 3.1). The **Misc** pulldown menu in the simulation tree window contains functions for that.

4. The primary *unfold options*. Here two options that are used during the unfold-process can be set. The **Unfold depth** denotes the maximal depth of the subtree that is generated by each **Unfold**-command. The **Narrowing power** option influences the maximal time that is spent during the evaluation of predicates; as our algorithm is only semi-decidable, a maximum value has to be supplied to prevent non-termination. A large number of other options can be set from the **Execute options** pulldown menu.

The relative sizes of the sub-windows can be changed by dragging (with the mouse) the □ grip between the simulation tree window and the specification window.

2.2 ADT Interface

The second major window is the Adt Interface, that contains a lot of functionality for the analysis of the abstract data type part of the specification. It is popped up by means of the **Adt Interface** command (in the **Misc** pull-down menu above the specification window), or by the **Instantiate** command in the **Event** pull-down menu (or the `F6`-function key). The layout of this window is depicted below, in figure 2.2. This window consists of four parts:

1. A *status line*, in which error messages and diagnostics are displayed. In the example, the line is blank.

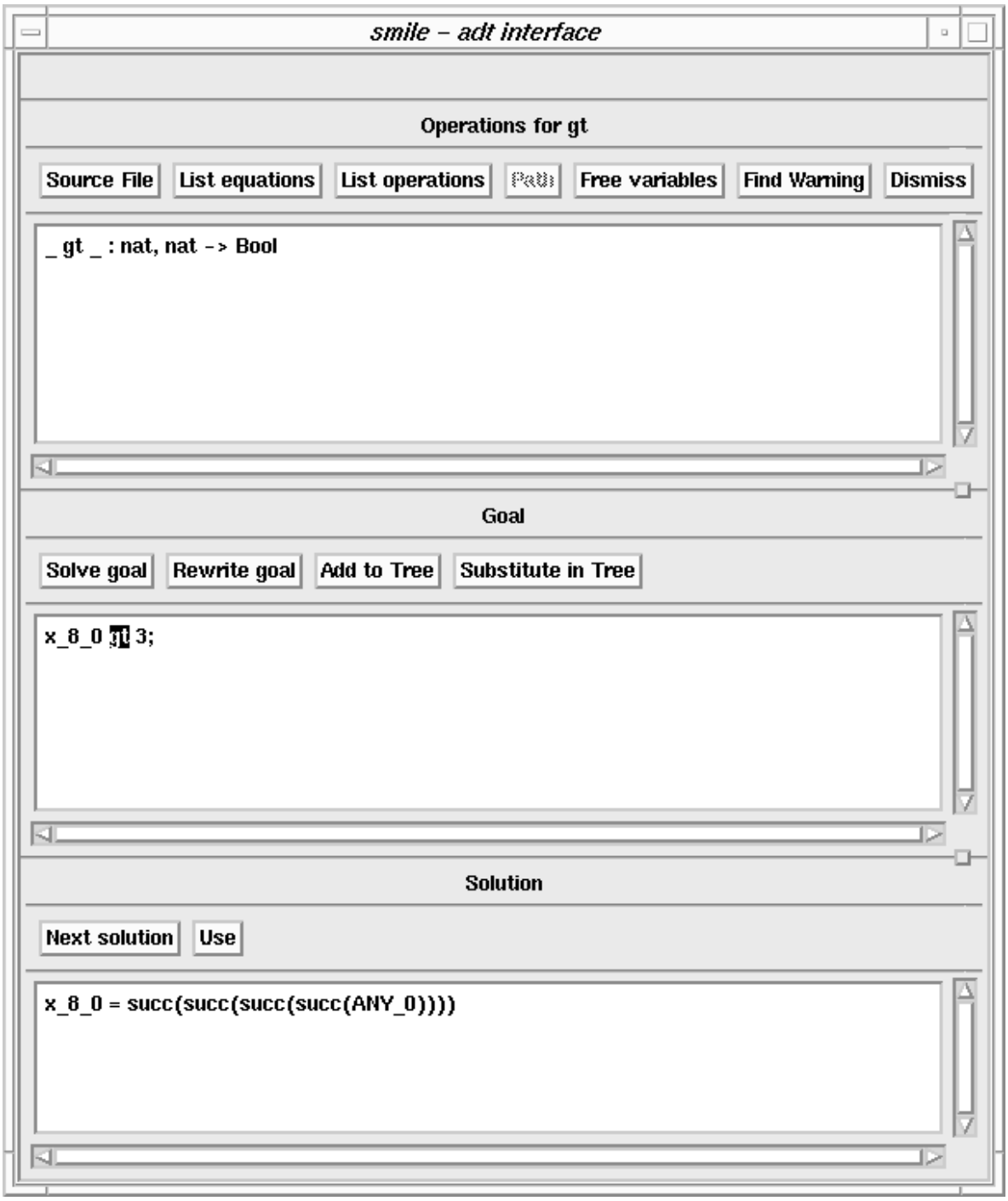


Figure 2.2: SMILE adt–window

2. The *source* window. In this window all kinds of information about the data type specification and about the instantiated event can be shown. In the example it contains the definition of the operator `gt`. This operator is selected in the *goal* window, after which the function **List operations** is called. If for instance the **List equations** button had been pressed, the window would contain

   ```
   x gt y = not ( x le y)
   ```

   The other functions are described in the reference manual. The **Path** button is disabled in this session. This is always the case if the adt–module is called using the **Adt Interface** command; if the **Instantiate** command is used, the button is enabled.

3. The *goal* window. In this window the goal, i.e. a number of predicates, can be typed for execution by the narrowing tool. Initially, this window is empty (when the **Adt Interface** is used to pop-up this window), or contains the predicates from the root node of the simulation tree to the selected event (when the **Instantiate** command was used in the main SMILE window). The **Add to Tree** and **Substitute in Tree** commands add the new goal to the selected event in the SMILE window. See section 4.3 for details.

4. The *solution* window. This window contains a solution for the goal typed in the *goal* window (if it can be found), or an error message otherwise. In the example, it contains the solution that `x_8_0` can be any natural number larger than, or equal to, 4. (The variables named `ANY_<int>` are generated by the narrowing tool). This window contains a solution for every free variable in the goal. Solutions can be added to the goal by pressing the **Use** button, after having selected a solution. **Next solution** generates the next possible solution. If it would have been pressed in the example, the *solution* window would contain the message 'All solutions found'.

2.3 Info window

A third window that is used by SMILE is the *Info window*. This window is used to display static information that is printed on request and can be saved to a file (e.g. the on-line help, information derived from the simulation tree, etc.). The layout of the info window is shown below, in figure 2.3.

The information in this window is not refreshed automatically by the tool. That is, if for instance the **Show Node** command is used to display the behaviour expression related to a certain state, and a new state is selected, the info window will not display automatically the behaviour expression that corresponds to the newly selected state.

On-line help is always displayed in the info window. The on-line help function is bound to the `Help`-key. The on-line help is context-sensitive; the displayed information depends on the area on which the mouse pointer is when the `Help`-key is pressed.

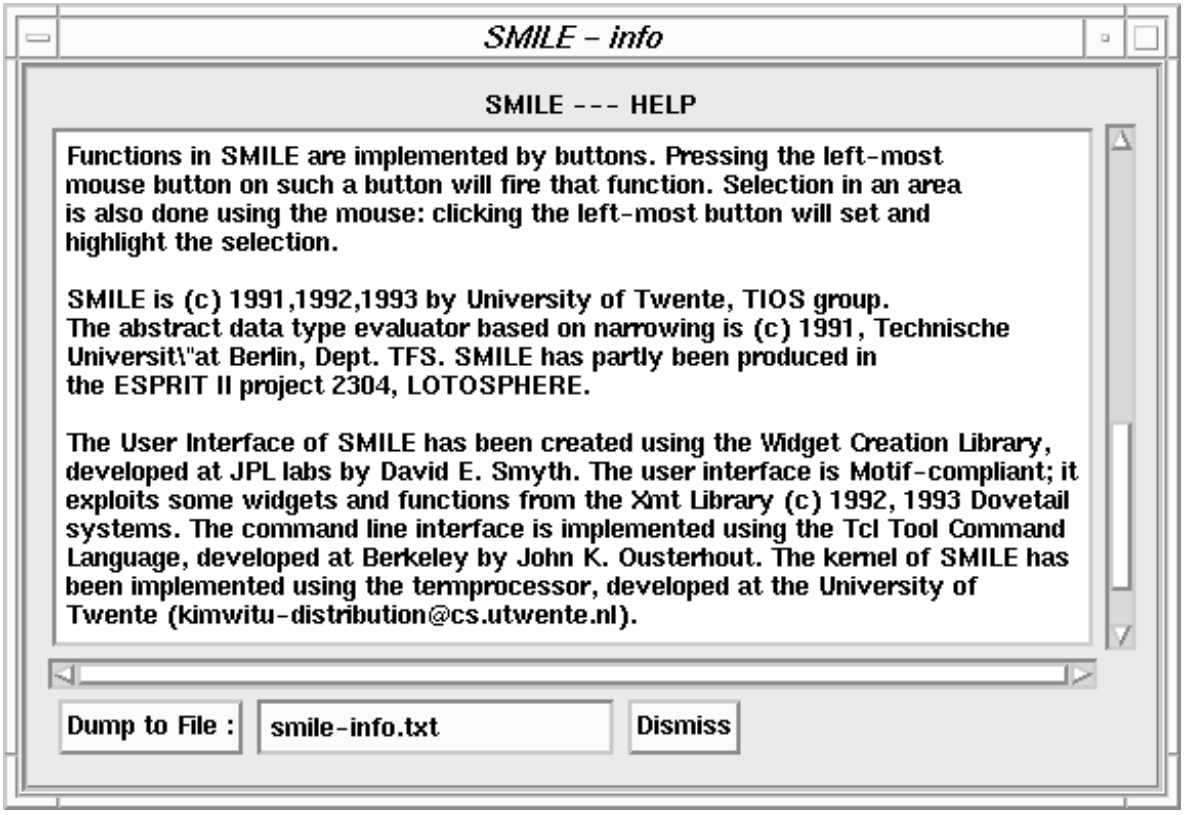

SMILE – info
SMILE --- HELP
Functions in SMILE are implemented by buttons. Pressing the left-most mouse button on such a button will fire that function. Selection in an area is also done using the mouse: clicking the left-most button will set and highlight the selection.
SMILE is (c) 1991,1992,1993 by University of Twente, TIOS group. The abstract data type evaluator based on narrowing is (c) 1991, Technische Universit\"at Berlin, Dept. TFS. SMILE has partly been produced in the ESPRIT II project 2304, LOTOSPHERE.
The User Interface of SMILE has been created using the Widget Creation Library, developed at JPL labs by David E. Smyth. The user interface is Motif-compliant; it exploits some widgets and functions from the Xmt Library (c) 1992, 1993 Dovetail systems. The command line interface is implemented using the Tcl Tool Command Language, developed at Berkeley by John K. Ousterhout. The kernel of SMILE has been implemented using the termprocessor, developed at the University of Twente (kimwitu-distribution@cs.utwente.nl).
Dump to File :
smile-info.txt
Dismiss

Figure 2.3: SMILE info–window

2.4 Command interface

It is also possible to *program* SMILE, using the Tool Command Language (Tcl; pronounce *tickle*), developed by John Ousterhout at Berkeley. The command interface of SMILE is popped up using the **Command Interface** command in the **Misc** pull down menu on top of the specification window. Programming SMILE is explained in detail in chapter 5. The window is displayed in figure 2.4, below.

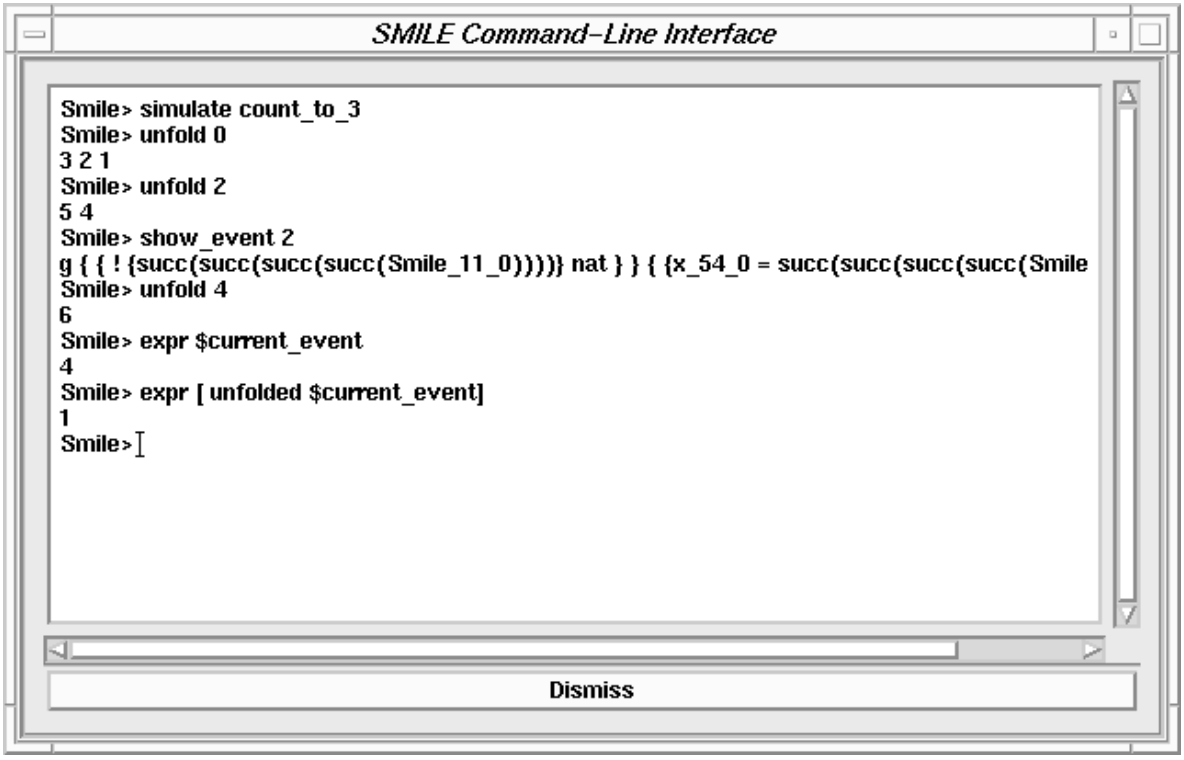


Figure 2.4: SMILE command–window

The series of commands depicted in that figure leads to the following simulation tree when our standard example is used:

```
<root-node>
    g !x_0
      (x_0 lt 3) = true
    g !succ(succ(succ(succ(Smile_11_0))))
      x_0 = succ(succ(succ(succ(Smile_11_0))))
        g !succ(succ(succ(succ(Smile_12_0))))
          Smile_11_0 = succ(Smile_12_0)
        g !succ(succ(succ(0)))
          Smile_11_0 = 0
            g !succ(succ(succ(0)))  (* Recursion detected *)
    g !succ(succ(succ(0)))
      x_0 = succ(succ(succ(0)))
```

As an arbitrary sequence of commands can be grouped into a procedure, using standard

control-structures like while-loops, if-then-else statements, etc., and because these commands can be bound to new buttons in the user interface of SMILE (see chapter 5), this allows one to enhance the functionality of the tool quite simply, and user-friendly.

Chapter 3

User Manual

This chapter will discuss some algorithms used in SMILE. First, the concepts of simulation trees and extended finite state machines (EFSMs) will be explained. In section 3.2 we will indicate how the evaluation strategy and the analysis of a simulation tree can be programmed using Tcl-commands. Subsequently, some special execution mechanisms will be discussed, that result in a partially generated tree. Partial evaluation basically amounts to generating only a part of the possible transitions. This is useful, if one is not interested in the complete simulation tree, either because it is too big to be analysed at once, or because only specific properties need to be analysed. SMILE offers, besides manually single stepping, five possible methods for that:

1. Execution of test cases. A test case is seen as an arbitrary LOTOS behaviour expression. Hence, it is possible to collect a number of test cases in one, large, LOTOS specification. SMILE then allows one to combine the test case with an arbitrary other part of the LOTOS specification. This is explained in subsection 3.3.
2. Supplying values for free variables. This is covered in section 3.4.
3. Ignoring behaviour expressions. This is covered in section 3.5.
4. Analysis of a particular part of the specification. Suppose we have a connection-oriented protocol in which some parts of the data transfer phase have been changed. Then one is usually particularly interested in that part of the specification. In section 3.6 we will describe an evaluation mechanism that allows one to specify a target state, and SMILE will subsequently produce a sequence of events that leads towards that particular target state.
5. Proving that a particular event exists. SMILE also allows one to specify a particular (series of) action(s). The tool then computes a trace that contains at least those specified actions. How that can be done, is explained in section 3.7.

Finally, some details about the narrowing algorithm are given, that might help to understand the behaviour of the tool in this respect. Also some hints with respect to a specification style of abstract data types are given there, that may result in more executable specifications. The material presented in this chapter is covered in more detail in [EW 92, BE 93, Eer 94].

3.1 Simulation trees and EFSMs

An (extended) finite state machine is a state machine in which the states are parameterised with some free variables. Of course, the transitions possible in those states also depend on the value of those free variables, and may update the free variables in the resulting state. SMILE computes such extended finite state machines, and uses that as basic data–structures.

A *simulation tree*, on the other hand, is the result of executing some transitions in such a state machine. That implies that, during execution of these transitions, the state instantiations are actually performed and it is checked whether all transitions in the state are possible with this particular initialisation.

Look for instance at the example specification, and suppose we simulate only the process `count_to_3`. The command **Complete EFSM** (in the **Execute** pull down menu), results in the computation of the state machine. This state machine can be written to a file by means of the command **Save EFSM as text** (in the **Misc** pulldown menu). This file contains then the specification (with the same name as the simulated process):

```
SPECIFICATION count_to_3 [g] (x_8_0: nat): noexit

(* EFSM generated by SMILE *)

(* EFSM corresponds to the full process definition *)


BEHAVIOUR
State1 [g] (x_8_0)
WHERE

    PROCESS State1 [g] (x_8_1:nat): noexit  :=
    [(x_8_1 eq 3)] -> g !x_8_1 ;
         State1 [g] (x_8_1)
    [] [(x_8_1 gt 3)] -> g !x_8_1 ;
         State1 [g] ((x_8_1 - 1))
    [] [(x_8_1 lt 3)] -> g !x_8_1 ;
         State1 [g] ((x_8_1 + 1))
    ENDPROC

ENDSPEC
```

Which is what would be expected, because the original process `count_to_3` is already in the format of an EFSM.

Now, if the entire specification is selected for simulation and the same procedure is used to generate the state machine, the result is

```
SPECIFICATION test [g]: noexit

(* EFSM generated by SMILE *)

(* EFSM corresponds to the full process definition *)
```

```
BEHAVIOUR
State1 [g] (0)
WHERE

    PROCESS State1 [g] (x_8_0:nat): noexit  :=
    [(x_8_0 eq 3)] -> g !x_8_0 ;
         State1 [g] (x_8_0)
    [] [(x_8_0 gt 3)] -> g !x_8_0 ;
         State1 [g] ((x_8_0 - 1))
    [] [(x_8_0 lt 3)] -> g !x_8_0 ;
         State1 [g] ((x_8_0 + 1))
    ENDPROC

ENDSPEC
```

The state machine `State1` is the same as in the previous specification, but only the initialisation of it is changed: the state machine is called with initial value 0, instead of with a variable.

Now we will execute this EFSM. Set the **unfold-depth** to 10 and give the **Unfold** command. The result is

```
<root-node>
    g !0
        g !succ(0)
            g !succ(succ(0))
                g !succ(succ(succ(0)))  (* Recursion detected *)
```

Apparently, SMILE has recognised that after having executed three transitions, it will never encounter any new behaviour. If the state machine is now printed again, it will look like

```
SPECIFICATION test [g]: noexit

(* EFSM generated by SMILE *)

(* EFSM corresponds to the full process definition *)


BEHAVIOUR
State1 [g] (0)
WHERE

    PROCESS State1 [g] (x_8_0:nat): noexit  :=
    [(x_8_0 eq 3)] -> g !x_8_0 ;
         State1 [g] (x_8_0)
    [] [(x_8_0 lt 3)] -> g !x_8_0 ;
         State1 [g] ((x_8_0 + 1))
    ENDPROC
```

```
ENDSPEC
```

where one of the transitions is deleted, because it has been proven, by exhaustive simulation, that it does not correspond to any event in the simulation tree, and is therefore not necessary to describe the semantics of the specification[Eer 94].

This illustrates the following: a simulation tree node always refers to exactly one state in the state machine; an event in the simulation tree always refers to exactly one transition in the state machine; a state in the state machine may relate to more than one node in the simulation tree (`State1` in the example relates to every node in the simulation tree); a state in the state machine may not have a corresponding node in the simulation tree (suppose that the transition with the predicate `x gt 3` would result in an invocation of a complex process `P`: then no transitions possible from `P` would ever occur in the simulation tree); a transition in the state machine may relate to any number of events in the simulation tree.

Two nodes in the *simulation tree* are identified only if

1. they relate to the same node in the state machine
2. the predicates from the root node to both simulation tree nodes are equivalent.
3. the values of the free variables computed for the node in the state machine are equivalent.

Equivalent means that the predicates (or the values) must be identical, up to renaming of variable names.

If during unfolding it is derived that the resulting node is equivalent to some other node, the event leading to the resulting node is annotated with `(* Analyzed elsewhere *)` or, if it is a loop, `(* Recursion detected *)`.

By default, SMILE generates the state machines automatically. But this implies that a lot of context information (values of free variables or constraints on free variables) is not used during the expansion itself. This may in some cases lead to a degrading performance. Therefore, it is possible to switch this feature off in the **Execute Options** pulldown menu, by means of the **Build EFSM as well** option. Note that when no EFSMs are generated it is possible that some states are expanded more than once. These kinds of information are displayed in the *statistics* window , that is displayed using the **Show statistics** command in the **Misc** pop-up menu on top of the simulation tree window. This window is displayed in figure 3.1.

The information in this window is subdivided into 5 groups:

States contains information about the number of states that has been generated. A state is basically the behaviour expression that corresponds to a node in the simulation tree. The value generated denotes the number of states that have been generated. reused expansions is increased whenever a simulation tree node is unfolded that corresponds to a state in the EFSM that has already been expanded (i.e. the transitions can be re-used). The field double expansions, on the other hand, is increased if a state is expanded

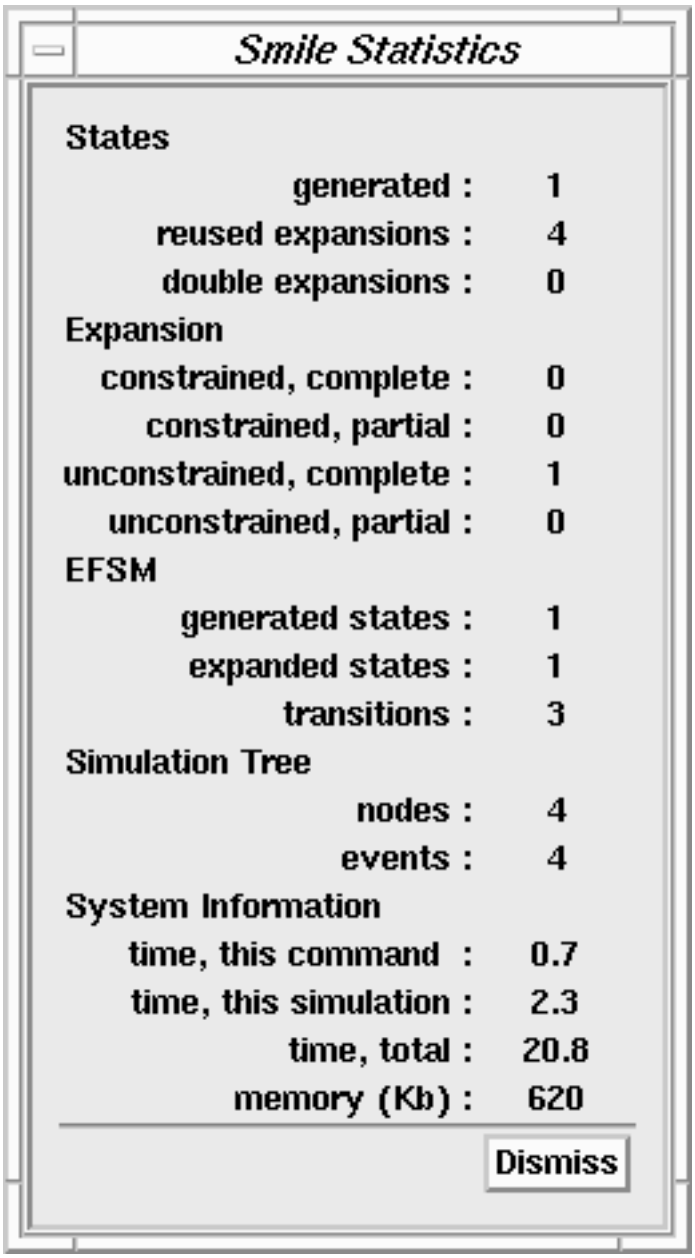


Figure 3.1: SMILE statistics window

for the second time; that is, the transitions that were initially produced could not be re-used, because one of the following reasons:

- Only a part of all possible transitions were generated. This happens when some behaviour expression has been ignored (using the **Ignore**-command); see also section 3.5. Partial expansions are also produced by the commands **Goto target** and **Next target**. The execute option **Filter Inference system** can be used to ensure that target-driven execution does not exploit any filters, and hence, always produces a complete evaluation.
- The expansion has used some initialisation values for the free variables in the state. This implies that for all other initialisations the state must be unfolded again. This is called *constrained* expansion (see below), and only happens when the **Build EFSM as well** option has been switched off.

Expansion contains information about the types of expansion that have been performed. *Complete* expansion is the default: a state is completely expanded if all components can be used (i.e. nothing is ignored), and no filter is applied. Otherwise, it is *partially* expanded. Also, *unconstrained* expansion is the default behaviour: this implies that no information about the values for the free variables of a state is used when expanding that particular state. The option **Build EFSM as well** switches this behaviour to *constrained* expansion, where all initialisations of state variables are used. This option is particularly useful if one is sure that a particular state in the EFSM is never (or seldomly) evaluated more than once, for instance because the specification is run in parallel with a test-case. This can easily be detected: in that particular case, the reused expansions counter in the **States**-section is 0. The option must be reset if the double expansions counter increases rapidly.

EFSM This section contains information about the size of the EFSM. If a state is part of the EFSM, and it is unfolded completely and unconstrained (default), then the generated transitions (and the resulting states of those transitions) will be added to the EFSM. In all other cases, the states and transitions will *not* be added to the EFSM. If the counter expanded states is equal to generated states, the EFSM is complete. In that case, writing the EFSM to a file will result in a specification that consists only of action prefix expressions, (generalized) choice expressions, process instantiations, and guards. If the EFSM is not complete, it can still be written to a file, but then the states that are not expanded will be represented by their corresponding behaviour expressions.

Simulation Tree The simulation tree section contains information about the size of the generated simulation tree. The counter nodes is *not* increased if a node is equivalent with some other node in the simulation tree, and is therefore annotated with either `(* Analyzed Elsewhere *)` or `(* Recursion Detected *)`.

System Information Contains information about the CPU usage and memory consumption of SMILE. The time, total counter contains the execution time (both system and user) in seconds of the current process. The time, this simulation contains the execution time of the current evaluation. It is reset whenever the simulation of a new behaviour expression is started. The time, this command counter contains the execution time since the last **Unfold**, **Goto Target**, **Next**, or **Complete EFSM** command has been issued. The

memory field contains an indication about the amount of memory comsumed by the *dynamically created* data structures of SMILE. This includes the size of simulation tree, EFSM, original specification, etc; but excludes the memory used for the window interface, for static (hash)tables, etc. The total amount of used memory can always be displayed using the Unix ps –v command.

The counters also indicate in which phase the tool is during the unfolding of a particular simulation tree node: First, the counter of generated transitions and states in the EFSM will be updated. If that has been done, one of the counters of performed expansions will be incremented. Then the transitions in the EFSM will be interpreted, resulting in events in the simulation tree (the event counter is incremented for each generated event). As this involves some checking of predicates, this might be time-consuming for large specifications. Finally, state matching is performed on the simulation tree, and the number of nodes in the simulation tree is updated. This phase is usually quite fast, except when the number of nodes that correspond to a single state is quite high (i.e. reused expansions or double expansions is high). In the latter case, a large number of time consuming comparisons and evaluations of value expressions must take place [Eer 94]. This is usually the case when a large part of the system control information is coded in the data part of the specification, instead of in the behavioural part.

3.2 Using Tcl to build a simulation tree

It is also possible to use the command-line interface of SMILE to start a simulation session, or to unfold a certain simulation tree node. The command-line interface contains an interpreter for the Tcl language[Ous 90, Ous 94]. To this language, we added a number of SMILE-specific commands. These commands do not work directly on events, or simulation tree nodes, but on an abstraction of that, viz. on so-called *event identifiers*. An event identifier is an integer value, that uniquely denotes an event (and implicitly the resulting node of that event) in the simulation tree. The special event identifier 0 is used to denote the root node of the simulation tree.

The basic commands that are added to Tcl are the following:

simulate *process* This command starts the simulation of the process that corresponds to process-identifier *process*. It corresponds to selecting that particular process with the mouse, and issuing the **Start**-command. The command returns no value.

unfold *event-identifier* This command returns a list of event identifiers that correspond to the children of the node that is denoted by *event-identifier*. If that node has not been unfolded, this will be done automatically.

unfolded *event-identifier* This command returns a boolean value that indicates whether the node corresponding to *event-identifier* has been unfolded.

father *event-identifier* This command returns the event identifier of the parent node of *event-identifier*. It is an error if *event-identifier* is 0.

show_event *event-identifier* This command returns a Tcl list of three elements: the first element is the gate identifier that corresponds to the event denoted by *event-identifier*, the second element is a list of value expressions that corresponds to the experiments of that event, and the third element is a list of conditions that belong to the event.

In chapter 6, we will give a complete overview of all commands that have been added to Tcl, and all variables that are shared between Tcl and SMILE. The major shared variable is `current_event`, that can be used to set and read the currently selected event in the X11-interface. In figure 2.4 we already gave an example using some of these commands.

These basic commands can be used to build and, more importantly, analyze a simulation tree. Notably on medium-size and large specifications, it is often not possible to analyze a simulation tree by hand. For instance, the following Tcl-program can be used to compute the path from the initial node to the give node:

```
proc path {end} {
  if {$end != 0}
  then {
    return [concat [path [father $end]] $end]
  }
  else {
    return $end
  }
}
```

where the square brackets `[` and `]` are used to denote the application of a Tcl-command, `$var` denotes the dereferencing of a variable, and `concat` is the builtin Tcl-command for list concatenation. This program can be typed immediately after the `Smile >`-prompt, or loaded automatically when starting up the tool (see also chapter 6).

For instance, checking whether an internal transition occurred on the path towards the currently selected event in the X11-interface can be programmed as follows:

```
proc has_gate {ev gate} {
   return  [ [ lindex [unparse_event $ev] 1 ] == $gate ]
}

proc count_internals {p} {
   set c 0
   foreach e $p {
      if [has_gate $e "i" ] then {
         incr $c
      }
   }
   return $c
}
```

```
expr [count_internals [path $current_event]] > 0
```

where the last line computes the desired result, using the previously defined commands `path` and `count_internals`.

Automatic unfolding of a complete simulation tree can similarly be programmed using the `unfold` command. A complete overview of the Tcl language, its syntax, and command structure is given in [Ous 94]. This book is available on-line, see the newsgroup `comp.lang.tcl` for details.

3.3 Executing test cases

A test case can be used quite conveniently to restrict the behaviour of the specification, viz. by executing the parallel composition of the test case and the specification. This can be done in SMILE by means of the **Run Test** command in the **Simulate** pull-down menu. This command pops-up a new window. This window is displayed in figure 3.2.

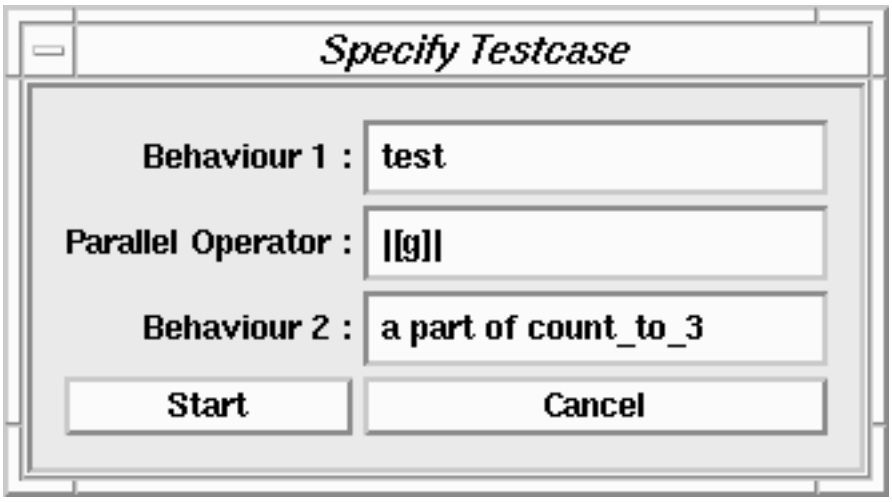


Figure 3.2: Specify test pop-up window

In this window three values must be given: two behaviour expressions and a parallel operator. Furthermore, two buttons are present: **Start** and **Cancel**. **Start** will start the simulation on the newly created behaviour expression. This is equivalent with having specified that particular behaviour expression in the original specification, selecting it, and pressing the **Simulate** button. The **Cancel** button pops the pop-up window down.

The behaviour expressions can be entered by selecting the appropriate behaviour expression in the specification window (leftmost mouse button), and pasting it into the text area (middle mouse button). Pressing the leftmost mouse button in the text area of the behaviour expression fields will highlight the corresponding behaviour expression in the specification window. Pressing the rightmost mouse button will delete the selection. The parallel operator may be any LOTOS parallel operator, and must be literally typed in the appropriate text area.

It is, in the current version, not possible to relate parameter values of both selected behaviour expressions directly. This can only be done indirectly, by instantiating the root node of the simulation tree (see section 3.4), and give identical values to the free variables of each component of the initial state. It is not possible to relabel the test case (or the behaviour expression) using the **Run Test** command.

SMILE currently contains no functionality to analyse the generated simulation tree automatically. If that is necessary (for instance, if a test case generates a huge tree with a lot of branching), it is possible to program the evaluation from within the command interface. Some examples are presented in chapter 5.

3.4 Instantiating variables

Instantiation of variables is a powerful technique that allows the SMILE user to play the rôle of the environment of the specification, by assigning a particular value to a variable that has been defined on an event. It is more powerful than test execution, because values can be assigned both to internally generated variables, and to variables that have been declared on externally visible events. Values can even be assigned to free variables of state expressions that are not visible in the simulation tree window. For instance, the LOTOS specification

```
choice x:nat [] i; g!x; stop
```

defines a value for `x` that is not observable. The state after the `i`-event already contains this value, and this value can be influenced by the SMILE user.

This works as follows:

The tool user first has to select an event in the simulation tree. Then he can use the **Instantiate**-function (in the **Event**-menu) to copy all conditions on the free variables of the resulting state of that event to the ADT-window. Here, he can play with these predicates, add restrictions, or assign values to predicates by adding equations like

```
variable = value
```

The free variables of the resulting state can be obtained by the **Free variables** command on top of the Adt window. These variables can be used to assign a value to. The actual instantiation takes place using the command **Add to tree** or **Substitute in tree**. Then all conditions and substitutions that are changed by the user are collected, and added to a *copy of* the selected event (with its subtree). This copied tree then reflects the events that are possible using the values supplied by the user.

It is also possible to assign values to the free variables of the root node of the simulation tree, by popping up the Adt interface using the **Adt interface** command in the **Misc** pulldown menu of the specification window. If then the **Add to tree** or **Substitute in tree** commands are used, the predicates are added to the root node of the simulation tree. In the reference manual the difference between the commands **Add to tree** and **Substitute in tree** is explained in more detail.

3.5 Ignoring behaviour expressions

In a large number of cases, a LOTOS specfication describes an arbitrary number of independent, identical, resources. For debugging purposes, it can be quite helpful to be able to restrict this number to, e.g. 1. That can be done either by rewriting the specification, or by means of the **Ignore** command. This command can be used to annotate an arbitrary behaviour expression. If this behaviour expression is then evaluated, it will *never* produce any transition: it is equivalent with **stop**. For instance, if we have a specification that is structured as follows:

```
    constraints [g] || resources [g] (s)
 where process resources[g] (s: set) :noexit :=
           [not(IsEmpty(s))] ->  choice x:e [] [ x IsIn s] ->
               (resource[g](x) ||| resources[g](Remove(x,s)) )
       endproc
```

Then ignoring the recursive instantiation of `resources` will result in the evaluation of the spec where it is assumed that there exists only one resource, for any non-empty set `s`. Another, quite popular, use of this command is in the selection of arbitrary constraints. In the same specification as above, the process `constraints` is usually defined as the interleaving of some independent constraints. The **Ignore** command can in these cases be used to switch arbitrary constraints on and off, while keeping all other constraints active. This is quite helpful when debugging constraint oriented specifications.

When a certain ignored behaviour expression is encountered when unfolding a node in the simulation tree, then it is possible that not all events are generated. The nodes that are only partially expanded contain a special event "..", that denotes that there might be more events possible from this node.

3.6 Execution towards a behaviour expression

In a large number of cases, one does not want to execute the complete specification, but merely wants to check that a certain state can be reached. For instance, in a connection oriented protocol, one may want to check whether it is in fact possible to set-up a connection. SMILE offers some functionalities for that. In the specification part of the main SMILE window, it is possible to select a behaviour expression (say a `DataRequest` event), and use this behaviour expression as a *target* behaviour expression. (Using the function **Set** in the **Target..** submenu of the **Simulate** pull-down menu.) If then the **Goto Target** command of the **Execute** pull-down menu in the simulation tree window is used, SMILE tries to execute the specification in such a way that the target behaviour expression is reached. ('Reaching' a behaviour expression means that the generated unfolding contains events produced by that behaviour expression). SMILE uses a specialized derivation system for that. This derivation system computes only those events that lead directly towards the given target behaviour expression (at least, it tries to compute those). The nodes are only partially unfolded. Hence,

the special event "`..`" is also displayed in the simulation tree window, to indicate that there might be more events possible from this node. The uninteresting parts are not evaluated (for instance: a choice expression has two operands. In a large number of cases, only one operand leads towards the target expression. In those cases, the algorithm will not evaluate the other operand). Therefore, a *partial* expansion takes place. This is also reflected in the **Expansion Info** part of the statistics window. For each expanded state it is indicated whether the expansion was complete or partial, and whether the environment has been taken into account or not (constrained; see the previous section). Note that partially evaluated states are not incorporated in the finite state machine. The analysis of which operands should be evaluated and which should not, is performed on the specification *text* only. That implies that values and synchronization(problem)s are not taken into account. This again means that the computation of next events may fail. This is indicated in the statistics window in the **double expansions** counter. If this happens quite often (and the **Build EFSM as well** option has not been switched off), it is probably more efficient to switch the **Filter inference system** execute-option off. This will result in a complete evaluation; the tool then uses the staticly derived information only to select the most promising next event, it will not use that information to perform partial evaluation. The event(s) resulting from the target behaviour expression are indicated in the simulation tree by means of an indication `>>`.

3.7 Proving that particular actions exist

3.7.1 Defining actions

Besides automatically executing towards a certain behaviour expression, SMILE is also able to validate whether a certain (sequence of) actions exists. Currently, the user interface to this functionality is rather poor, and exists only in the command-line interface. An action *ac* is defined by the following syntax:

ac	=	*gate* [*exp* LIST] ["[" *cond* CHAIN ";" "]"]
ac	=	*gate* "∗"
gate	=	*g* | "*i*" | "`exit`"
exp	=	"!" *value–expression*
cond	=	*value–expression* "=" *value–expression*

Where a *value–epression* is an arbitrary value expression, and *gate* is an arbitrary gate identifier. It is possible to use new variables in the actions, as long as it is always possible to map the first occurrence of a variable onto a unique sort name.

This implies that the following expressions can be used as actions:

```
1.   g
2.   g !x of int
3.   g !y of int
4.   g !3*x !x
5.   exit !x of int
6.   g !x [x lt 4; x gt z]
```

```
7.   i !x [x gt 3]
8.   g *
```

This shows that the actions that SMILE accepts as target expressions are more powerful than the LOTOS action denotations: it is possible to use values on internal events. The special value `*` denotes an arbitrary number of arbitrary values. Hence, the last example matches the actions 1, 2, 3, 4, and 6.

The commands that can be used in the command-line interface of SMILE, however, do not operate directly on these actions. They first must be parsed into an *event identifier* by means of the command `parse_action`. This command returns an integer value: the event identifier that corresponds to the given action.

3.7.2 Using actions to execute a specification

SMILE tries to find *overlapping* events for the given actions. An event is defined to be overlapping with an action if it may synchronize with it using the LOTOS synchronization rules (in this special case, we assume that `i` behaves like any other gate identifier).

The function that locates a particular action is `goto_target`, that is defined as follows:

goto_target *target-description event-identifier* This command starts the evaluation at the node indicated by *event-identifier*, and returns a list of event-identifiers. Each generated event identifier has the same parent node (usually, only one event identifier is being generated), and conforms to the property that the *target-description* has contributed to that event. For an action description, this means that the event overlaps with the given action description.

A target-description *td* is more general than a simple action: it conforms to the following syntax:

```
            td :  simple-target LIST
             ;
simple-target :  process-identifier
             |   integer LIST
             ;
```

where LIST denotes a Tcl-list (i.e. either a singular element, or a number of elements enclosed in braces '{' and '}'. Each integer corresponds to a parsed action description (i.e. it is possible to re-use a parsed action), and a process identifier corresponds to the same evaluation mechanism as outlined in the previous section. The list of integers denotes a target that consists of any of the specified actions.

For instance, a sequence of 3 arbitrary actions labelled with `g` are derived as follows:

```
Smile> set action_name [parse_action "g *"]
Smile> goto_target { $action_name $action_name $action_name} $current_event
```

```
28
Smile>
```

The `goto_target` command in this fragment results in a simulation tree in which the event denoted by event-identifier 28 is reached by a trace containing tree events labelled with `g` from `$current_event`. Note that the target description is a list of three elements, whereas each element is a singular list. The following target expression (consisting of a target expression of which the first element is a list of two elements) can be used to locate an event that is either labelled with `g` or `h`, starting from the initial node of the simulation tree:

```
Smile> goto_target { {[parse_action "g *"] [parse_action "h *"] } }  0
17
Smile>
```

Note that this evaluation mechanism is only useful if the simulation tree has not been unfolded yet. If the simulation tree has already been unfolded, it is much more efficient to program this using analysis functions over the simulation tree.

The function `goto_target` computes only a few (usually, only one) event-identifier(s) that correspond to the given target expression. Sometimes, one may be interested in more than one of these events. The command `next` can be used to compute another (list of) event-identifier(s) that also correspond to the last given target expression:

next This command can be used to continue the evaluation after the last `goto_target` or `next` command, and produces a list of event identifiers. Each produced event identifier has the same parent node, and corresponds to the last simple target description in the target that has been used in the last `goto_target` command.

If the simulation tree is finite, this implies that, for instance, all first event identifiers can be found that correspond to a given action. The next Tcl-procedure computes such a complete list of events:

```
proc find_all_sequences {action node } {
  set result [ goto_target [parse_action $action] $node]
  set new  $result
  while {[llength $new] >0 } {
   set new [next]
   set result [concat $result $new]
  }
  return $result
}
```

Hence, the command `find_all_sequences "g *" $current_event` will generate all traces that start at `$current_event` and end with `g`, and do not contain any intermediate `g`-actions (the algorithm stops at the node where the target has been reached; it will never examine any events after such a node).

The procedure `find_all_sequences` can easily be changed such that it also works on infinite simulation trees, by adding an (optional) limit to the length of `result`.

3.7.3 How does this work

The `goto_target` command basically does two things:

1. It analyses each node in a collection of nodes N (initially, only the node corresponding to the parameter of `goto_target` is in N), and tries to find out how the target can be reached from a node, using only its control structure (i.e. it abstracts from the concrete data values). On the basis of that information, a certain cost is assigned to each node (see below), and the tool selects the node with the lowest cost.

2. After a particular node has been selected for evaluation, this node is removed from N, and unfolded using the information computed in the first phase. This computation results in only a part of all possible events (only the necessary events are generated). It is checked whether the target expression corresponds to the produced events. If that is the case, the algorithm successfully returns with these events (that is the reason that the produced events all correspond to a single node). The event-identifiers of the events that do not correspond to the target expression are added to N, and the process continues.

The `next` command simply continues this computation with the remaining elements in N.

The effectiveness of this procedure can be influenced in 3 different ways:

- Not filtering the inference system. This results in the computation of all possible events in phase 2. This is sometimes useful, because the tool is not always able to produce all interesting events. The tool is able to detect this, and if that is done, the node is re-inserted in the set N after the events are produced, and marked such that when it is selected again, it will be completely evaluated (instead of only partially). If this happens often (this can be detected in the SMILE Statistics window), filtering should be switched of to obtain a better performance. This can be done using the **Filter inference system** execute-option in the simulation tree window, or by setting the boolean Tcl-variable `$partially_evaluate`. The variable corresponds directly to the button (i.e. setting the X11-toggle button will update the variable, and vice-versa).

- Influencing the cost function. The cost function is based on analysis of the filters, and counts the minimal number of action prefix expressions that must be evaluated before the goal expression is reached. Furthermore, it may also contain a recursion operator (that represents the possibility that a LOTOS process is evaluated again to obtain a new transition using new values). Each recursion-operator also plays a rôle in the cost function, because the probability that a necessary event is actually inferred from a recursive instantiation is lower than the probability that it is inferred from a node that has never been analysed earlier. The Tcl-variable `recursion_cost` can be used to set this value. Default value is 1 (i.e. recursion counts as one additional event).

Another component in the cost-function is the number of times a state corresponding to a particular node has already been used earlier (in another simulation tree node). This value is multiplied with the value of the Tcl-variable `reanalyze_cost`; its default value is 3. Finally, it is possible to defer the analysis of nodes that have already been analysed earlier, but might not have produced all necessary results. This is useful, because the check that has been implemented in SMILE is rather weak; hence, in a large number of cases this re-analysis does not produce any necessary events. The default behaviour of SMILE is to prefer always a node that has not been analysed earlier above a node that has been analysed, but whose evaluation resulted in this exception. The boolean Tcl-variable `postpone_reevaluation` can be reset if this behaviour is not wanted. Note that in all cases the cost of a re-evaluated node will increase with `$reanalyze_cost`, as the (EFSM)-state will effectively be re-used.

- Changing the evaluation strategy. It is still possible that the cost function does not discriminate enough, such that more than one node has the same cost. In that case, the node is selected that is inserted first in the set N (if the evaluation strategy is breadth-first), or last (if it is depth-first). The evaluation strategy is set using the boolean Tcl-variable `breadth-first`. (If the value of the variable is False, the strategy is depth-first). This variable is connected to the **Execute option** titled **Breadth-first** in the simulation tree window.

Note that this functionality is only helpful in proving that a certain action is possible, or that a behaviour expression is used. It does not do any miracles with respect to state space exploration: for specifications that cannot be represented by finite EFSMs, it might even be the case that the algorithm does not terminate at all, although a successful event may exist. Use the **Stop**-command if you think that evaluation takes too much time.

Best results are obtained on specifications in which the major control is in the behavioural part, and not in the data types, and with *simple targets* (cf. the syntax of target expressions) that result in traces of, say, maximal 10 events. If you are almost sure that the generated trace will consist of more than 7 or 8 events, it is usually more efficient to split the target expression into several sub-targets, and use a list of these sub-targets instead of a single simple-target.

3.8 Narrowing

SMILE incorporates a narrowing algorithm to compute whether a collection of predicates has a solution or not. This algorithm works on the equations of the ADT specification. This algorithm heavily depends on a strong distinction between constructor terms and functions. *Functions* are operations that occur on the outermost position of the lefthandside of an equation. *Constructors* are operations that occur at outermost positions of sub-terms of the lefthandside of an equation (i.e. they describe the pattern on which the function should operate). The algorithm works by assigning values to free variables in the predicate by unifying a term in the predicate with the left-hand side of an equation in the specification. Then it assumes that the resulting values for the free variable(s) in the predicate are constructor terms. Hence, the tool is not able to handle ADT specifications efficiently if the same operation is used both as a function and as a constructor. Furthermore, it should be clear which equation

should be applied to a function term. Therefore, execution of specifications in SMILE is most efficient if the abstract data type part is constructive, and if no overlapping equations occur(that is: two or more equations can be applied to one function-application).

The tool incorporates algorithms that check whether these requirements are met. The specification text in the ADT-interface window (use the **Source File** command in the Specification part of the Adt window) is interspersed with warnings concerning the executability. The following warnings may occur:

1. **Constructor ... also used as function**. This message is very serious. It indicates a violation of the constructiveness of the specification. It is given near an equation, in which one of the constructors is also used as a function somewhere else. The exact function can be found by clicking each operation symbol consecutively, and pressing the **List equations** button, to see whether any equations are given for that particular operation. Solving this problem will lead to much better completeness of the narrowing algorithm, and therefore in a smaller state space.

2. **Equation overlaps with another equation**. This message indicates that it is possible to apply two different equations to the same value expression. Therefore, it depends on the strategy of the tool which equation is used, and this might lead to unnecessary non-termination of the algorithm. The tool incorporates some heuristics: it applies the easiest equation first (measured by the complexity of the lefthandside of the equation). But of course, this heuristic may fail....

3. **Equation may overlap with another equation**. In this case the tool was not able to prove whether the equation overlaps or not. In most of the times, this has to do with not being able to prove whether two conditions for equations may hold together or not.

4. **Equations for this function may be nonterminating.** This indicates possible non-termination of evaluation with these function. It contains a *maybe* clause, because the tool applies narrowing to evaluate the righthandside of the equation. As narrowing is semi-decidable, it may be the case that the equations in fact *do* terminate. But if you know that the sorts on which the equations are defined are finite (for instance service primitives, or some other enumeration types), then the warning is to be taken very serious.

5. **This constructor is not used in any equation.** This either occurs because no functions are defined over that particular type, or because the constructor has been omitted from all equations (for instance because it was added later and the functions were not updated accordingly).

6. **Constructors ... missing in equations for this function.** This indicates that this particular function is only partially defined. If that is intended, there is of course no problem at all. But if the function is used as a total function, the tool will produce unexpected evaluation results.

The narrowing algorithm is completely based on the equations in the abstract data type specification. So if the functions are not completely specified, the tool will not be able to handle the specification as you might have expected.

More information about the implementation of narrowing, and an operational description of the algorithm can be found in [EW 92].

Chapter 4

Reference manual

This chapter describes the commands of SMILE. All commands are given by the user by clicking the leftmost mouse button on one of the command-buttons on the screen or in a pull-down menu. Some commands are also bound to a (function) key. Where appropriate, this is indicates at the command description. The parameter of a the command is either the selection in the appropriate window (which can always be displayed by clicking the middle mouse button in that window) or given in the control panel. We list the commands here in the order they appear on the window: First we describe the commands on the SMILE specification window, then the commands in the Simulation tree window, and finally the commands in the ADT window. In the next chapter, we will describe the commands and variables that have been added to Tcl, and can be used in the SMILE command window.

In the sequel, the phrase 'selected node' refers to resulting simulation tree node of the selected event in the simulation tree window, or to the root node if the string `<root_node>` is selected.

4.1 SMILE Specification window

4.1.1 Simulate pull-down menu

Simulate

Key binding: The `Enter`-key, when pressed in the specification window.

Parameters: The selection in the specification window.

Result: The current simulation tree in the simulation window is deleted. The selected behaviour expression or process is transformed into a state. This state becomes the root-node of the simulation tree. The name of the selected process is displayed in the title bar of the simulation tree window. If the selection was not a process definition but a behaviour expression, the title bar contains the phrase "Simulation Tree of a part of *pid*", where *pid* is the process identifier of the process in which the behaviour expression is defined.

The selected behaviour is highlighted in the specification window.

Run Test

Parameters: None

Result: The specify test pop-up window appears. A test can be specified and executed. This is discussed in more detail in section 3.3.

Quit

Key binding: `CTRL-C` when pressed in the specification window.

Result: Stop the current expansion and exits the simulator. Can be used any time, even during start-up.

Target.. sub-menu

Set

Key binding: `CTRL-T` (when pressed in the specification window).

Parameters: The currently selected behaviour expression in the specification window.

Result: The currently selected behaviour expression becomes the new target behaviour expression. See also section 3.6, and the commands **Goto Target** and **Next Target**.

Show

Key binding: `CTRL-F` (when pressed in the specification window).

Result: Show the current target behaviour expression.

Reset

Key binding: `CTRL-R` (when pressed in the specification window).

Result: Clear the target behaviour expression. This command disables the **Goto Target** command.

Ignore.. sub-menu

Set

Key binding: `CTRL-I` (when pressed in the specification window).

Parameters: The currently selected behaviour expression in the specification window.

Result: The currently selected behaviour expression is ignored during unfolding (see also the previous chapter). Note that more than one behaviour expression can be ignored at the same time. An ignored behaviour expression is treated in the same way as `stop`: no events are produced, and, hence, it is not able to synchronise with any ignored behaviour expression.

Show

Key binding: `CTRL-S` (when pressed in the specification window).

Result: Show the next ignored behaviour expression.

Reset

Key binding: `CTRL-D` (when pressed in the specification window).

Parameters: The currently selected behaviour expression in the specification window.

Result: Use the currently selected behaviour expression again. This operation is the inverse of **Set**. It is equivalent to a no-op if the currently selected behaviour expression is not ignored.

4.1.2 **Options** pull-down menu

Show behaviour

If this option is set, the behaviour expressions of the processes are shown. This is especially useful when the **Find action prefix** command is used. The default is that only the headers are shown, which allows a quick browse through the list of processes.

Show extensions

During the static semantics checking phase, each value identifier is extended such that it gets a unique name. Normally, these extensions are not shown. This may lead to some confusion when trying to solve the predicates. Take for instance this correct LOTOS behaviour expression:

```
choice x: nat [] [x=4] -> g? x: nat [x=3] ; stop
```

This will lead to an expansion which looks like:

```
g ? x_1 : nat
    x_1 = 4;
    x_1 = 3;
```

Now, one could wonder why the narrowing tool is not able to resolve this predicate. In such a case, the extended identifiers unparsing option should be used. This produces a specification

```
choice x_4: nat [] [x_4 = 4] -> g? x_5: nat [ x_5=3] ; stop
```

and a simulation tree

```
g ? x_5_1 : nat
     x_4_1 = 4;
     x_5_1 = 3;
```

which makes the result immediately clear. The extensions `_5` and `_4` are arbitrarily chosen here.

Setting this option forces a redisplay of both the specification window and the simulation tree window. This toggle option also appears in the **Display** pull-down menu in the simulation tree window.

4.1.3 **Misc** pull-down menu

Adt Interface

The Adt Interface window is popped-up. The 'goal' window is cleared, and the **Path** command will not work there.

Command Interface

The SMILE Command window is popped-up. Any Tcl-command can be typed there. In that window, commands can be given to analyse the simulation tree, or to use special execution mechanisms. See also a previous section on locating action denotations, and the next chapter for an overview of the additional SMILE-specific commands.

4.1.4 **Help** pull-down menu

Help

Key binding: Pressing the **Help** key on any area or button in any window of SMILE will give specific help on that area or command.

Result: The info window will pop-up and an appropriate help message will be printed in that window.

4.2 SMILE **Simulation Tree Window**

4.2.1 **Execute** pull-down menu

Unfold

Key binding: The `Enter` key.

Parameters: The selected node in the simulation tree.

Options: There are a number of special options in the **Execute options** menu. The effect of these options will be described in the next subsection. Two options can be set in the main SMILE window: the **Unfold depth** and the **Narrowing power**.

The **Unfold depth** can be used to set the maximal depth of the generated tree. That is how automatic execution is performed: setting the unfold depth to an extroardinary value (say, 100000) will allow you to go to lunch, and when you return, SMILE has either still busy computing a simulation tree, or has computed a complete tree. The default value is 1.

The *Narrowing power* is a measure of the amount of time the tool is allowed to spend when it evaluates each goal during the unfolding of a state, or the creation of an EFSM. When set to 0, narrowing is switched off. The default value is 2. The option influences the narrowing tool, by setting the maximum number of narrowing steps and the maximum number of instantiations. The number of narrowing steps is $2000 *$ narrowing power; the maximum number of instantiations is set to $25 *$ narrowing power. Increasing this value means that more values for the free variables are tried by the narrowing tool, before it runs out of bounds. If the narrowing tool cannot find a solution for the predicates corresponding to a certain event in the simulation tree, the events will be prepended with a '–' sign. If checking those equations in the narrowing interface produces solutions (see the **Instantiate** command below), it might be useful to increase the narrowing power. But if the predicates simply cannot be solved, increasing the narrowing power simply implies that it takes more time to find that they cannot be solved, so some performance is actually lost in that case. For large specifications with complicated data types, the default value should be increased to, say, 20, depending on the specification. Please experiment with this value, until the most optimal results are obtained.

Result: Note that we assume that the Execute options have their default value. The effects of these options are described in the next subsection.

The selected simulation tree will be unfolded up to the given depth. This results in a simulation tree, that consists of a number of *events*, that are possibly annotated. The events are displayed using additional indentation. If the first 'event' is the string "..", this implies that the unfolding was only partial. This occurs if some ignored behaviour expression (cf. the **Ignore** command) has been encountered.

Unfolding proceeds as follows: first it is checked whether the state in the state-graph corresponding to the to-be-unfolded node in the simulation tree is already expanded. As explained in the previous chapter, this may be the case even if the node in the simulation tree is not marked. If it has already been expanded, the counter **reused expansions** will be increased. Otherwise, the state in the state graph will be expanded first, and one of the counters below **expansion** will be increased.
Then for each possible transition in the state-graph it is checked whether that particular transition is possible from the current node in the simulation tree. This implies that the way in which the current node in the simulation tree is reached is taken into account. Progress is again indicated in the control panel: the number of events in the simulation tree is being increased. Finally, all nodes that can be reached by the new events are checked whether they match other, already generated, nodes. If this is not the case, the number **nodes** in the **Simulation Tree** section is increased.

The tree is built using a breadth-first evaluation strategy. This implies that first all events at depth 1 will be generated, then all nodes resulting from these events are evaluated, etc. A node is not evaluated if it is annotated with either recursion detected, or analyzed elsewhere (see below).

The generated events can be annotated as follows:

(* Recursion detected *) This means that the resulting node of this event is identical to some other node that is on the same path to the root node.

(* Analyzed elsewhere *) This means that the resulting node of this event is identical to some other node that is not on the same path to the root node.

– (minus sign) This means that the tool is not able to prove whether there exists a solution for the predicates that are associated with the collection of events from the root node to the current event. Increasing the narrowing power might help.

The first two annotations are listed after the gate and experiment, but before the list of predicates; the third annotation is printed in front of the event.

Diagnostics: The current depth is displayed in the error line. If the maximal depth is reached the message "Unfold: ready after ... steps" is printed.

If a non-guardedly well-defined specification is being simulated, a warning message 'Specification not guardedly well-defined' is printed in the error line. Not guardedly well-defined means that a recursive call to a process is encountered before the process executed an action prefix expression. In that case, it is possible that the resulting simulation tree contains only a subset of the behaviour which is actually possible, because every possible event contributed by the non-guarded recursive call is not there.

This means that the resulting EFSM is probably not complete as well. See also the # **unguarded recursion** execute option.

One step

Key Bindings: This function is bound to the right-arrow cursor key (usually labelled with →). This key-binding is only active when the mouse-cursor is in the simulation-tree window.

Parameter: The currently selected node.

Result: The currently selected node is unfolded, but only one step. This function is usually used only with the key-binding.

Goto target

Key Bindings: This function is bound to the `CTRL-G` key (and is only active when the mouse cursor is in the simulation tree window).

Parameters: The currently selected node, and the currently selected target behaviour expression (see the **Target** pull-down menu, described in the previous subsection).

Result: See section 3.6. Basically, the selected node is partially unfolded up to an arbitrary unfold depth, until the selected target behaviour expression is active. This might result in unfoldings that contain the string "..". This string denotes that only a part of all possible events have been computed. During unfolding, the value of the cost function (cf. section 3.7) is displayed in the Message window ('Target at distance n'). If the target expression has contributed to a particular event, this is indicated with the string >> in front of that particular event. The function stops then, and prints the message "Target Reached". If the target expression cannot be reached from the selected node, this is reported by the message "Target cannot be reached".

Next Target

Key Bindings: This function is bound to the `CTRL-N` key (and is only active when the mouse cursor is in the simulation tree window).

Parameters: None.

Result: This function produces the next (set of) event(s) in which the target expression has been active. It can only be used after the function **Goto target** has produced a (set of) event(s) that corresponds to the target expression. It is the same as the Tcl `next` command, described in section 3.7. The result of this function is similar to that of **Goto target**, described above.

Until stable

Key Bindings: This function is bound to the `CTRL-S` key (and is only active when the mouse cursor is in the simulation tree window).

Parameters: The currently selected node and the **Unfold depth**.

Result: This function unfolds the subtree below the parameter node, until each node is stable. A node is stable, if it contains no outgoing events labelled with `i`. The maximum length of the sequence of internal events that is executed can be set by the **Unfold depth**.

Complete EFSM

Parameters The narrowing power (see **Unfold**).

Result The finite state machine corresponding to the currently simulated behaviour expression is made complete. That means that every state in the state machine is expanded. Progress is indicated in the control-window, in the section titled 'State-Graph Info'. There the number of generated states and the number of expanded states is printed. Additionally, also the number of generated transitions is printed there.

It is not always possible to derive a finite state representation of a LOTOS behaviour expression. Some specifications simply do not have a finite representation, because they exploit one of the following LOTOS constructs: recursion over the parallel operator, recursion in the lefthand side of the enable or disable operator. If these recursive calls do not exist (and enough resources are available), a finite state machine will be produced by SMILE.

Stop

Key Bindings: This function is bound to the `Break` key (on some keyboards, this key is labelled with `Pause`).

Result: Any unfolding that is currently busy is interrupted and stopped.

4.2.2 **Execute options** pull-down menu

These options are active during the unfolding of a particular node, or when generating an EFSM. Except for the **# unguarded recursion** option, they all correspond to boolean values, and can therefore only be set or reset. When an option is set, this is indicated with a □-mark in front of it.

The pull-down menu contains the following entries:

Rewrite Always

Default: False

Effect: Rewrite all value expressions in the events into their normal form.

Rewrite Minimal

Default: True

Effect: Rewrite the value expressions only if the result is smaller (in terms of the number of operation symbols) than the original. This option is ignored if **Rewrite Always** is set.

Disable state normalization

Default: False

Effect: By default, each generated state is normalized according to strong bisimulation congruence laws. Setting this option disables this normalization.

Filter Inference system

Default: True

Effect: This option is only used during target-driven execution. If set, it specifies that only the interesting events will be generated. This may result in a large number of re-evaluations. In those cases, this option can be switched off. See also section 3.6 and section 3.7.

Substitute Unique solutions

Default: True

Effect: If set, the tool tries to find a unique solution for the set of predicates. If such a solution exists, the solution will be used instead of the predicate. This results in better state matching. Setting this option implies that new variables can be introduced. These are named 'Smile_¡int¿', and denote arbitrary values of the appropriate type.

Build EFSM as well

Default: True

Effect: If set, the tool will automatically build an extended finite state machine when unfolding behaviour expressions. This results in a better performance if the number of states that is visited more than once is rather high. Unsetting this will result in a better performance if the specification uses a lot of parameters that are bound to ground terms instead of to arbitrary variables. If this option is not set, it is not possible to write an EFSM (see **Write EFSM**).

Breadth-first

Default: True

Effect: This switches the unfold-strategy from breadth-first to depth-first. 'Breadth-first' means that all children of the current event are evaluated first, then all grandchildren, etc. Depth-first, on the other hand, means that first the first child is evaluated, then the first child of that child, etc. This option is particularly important when doing target-driven execution, as in that case only a part of the tree is generated.

Trace behaviour expressions

Default: False

Effect: If set, every behaviour expression that is evaluated is highlighted for 0.5 seconds. This allows one to trace the evaluation of the tool in the specification window.

 # Unguarded Recursion

Default: 0

Effect: Here one may set the number of times an unguarded recursive call must be executed. Default is 0; i.e. an unguarded recursive call does not result in any event. If this limit is exceeded, an error 'Specification not guardedly well-defined' will be given (see also **Unfold**).

4.2.3 **Event** pull-down menu

The commands in this menu have one parameter: the selected event in the simulation tree window.

Hide

Key Bindings: This function is bound to the `F2` key.

Result: The selected event in the simulation tree and its corresponding result-node are hidden. This is indicated by the string '`(* .. hidden event .. *)`' in the simulation tree window. Hidden events will not be unfolded. This function is especially useful if the narrowing tool is not able to prove that the conjunction of certain predicates is false. Hiding the event then allows a user to continue the simulation, but without being bothered by the unreachable subtree.

If the selected event was already hidden, this function reveals the event again. If two sibling events are hidden, only one indication `(* .. hidden events .. *)` will appear at the beginning of that event list. If the **Hide** command is executed on this indication, all hidden events will reappear.

Hidden events are a part of the finite state machine. They are *only* hidden in the user-interface; the generation of finite state machines completely relies on the power of the inference system and the narrowing engine, it cannot be influenced by the user (except by setting the narrowing power).

Hide rest

Key Bindings: This function is bound to the `F3` key.

Result: All events in the simulation tree are hidden, except the events on the path towards the currently selected event. This is useful if a large subtree has been generated, but one is only interested in one single trace. Then this function will remove the other evenst from the screen. There is no easy way to un-do this function: every hidden event can only individually be revealed (using the **Hide**-function).

Find action prefix

Key Bindings: This function is bound to the `F4` key.

Result: The action prefix expression(s) which contributed to the selected event are highlighted in the specification window. If only the process headers are shown, the process in which the action prefix expression is defined is highlighted. Pressing the button again shows the next action prefix expression. The bell is rung when the last action prefix expression was highlighted.

Rewrite event

Key Bindings: The `F5`-key.

Parameters: The currently selected event.

Result: The value expressions in the selected event will be rewritten into their normal form.

Instantiate

Key Bindings: This function is bound to the `F6` key.

Result: All predicates from the root-node to the selected event are collected and printed in the 'goal' sub-window of the 'Adt Interface' window. The first solution for these predicates is printed in the 'solution' part of that window. This allows the analysis of the predicates, and also enables the SMILE user to add constraints to the event (like giving values for certain variables). These new values can be added as new constraints in the simulation tree using the command **Add to Tree** or **Substitute in Tree** in the Adt Interface (see also sections 3.4 and 4.3).

4.2.4 **Node** pull-down menu

The entries in this menu use the resulting node of the currently selected event as parameter, or the root node of the simulation tree if the string `<root-node>` is selected.

Identical node

Key Bindings: This function is bound to the `F7` key.

Result: If the simulation tree node has been analysed elsewhere (an indication is added to it), then the event leading to the identical node becomes the currently selected event.

Diagnostics If the node has not been analysed elsewhere an error message is given.

Show menu

Result: The initial events resulting from the currently selected node are shown in the info window. If the selected node is not unfolded, nothing will be shown.

Show path

Result: The path from the root node of the simulation tree to the currently selected event is printed in the info window.

Show node

Result: The behaviour expression that corresponds to the currently selected node is printed in the info window, together with the computed values for the free variables in that behaviour expression, and the predicates on these values.

4.2.5 **Display** pull-down menu

This pull-down menu contains 3 options, that can be set by clicking on them.

Show predicates

Default: True.

Effect: If set to False, the predicates in the simulation tree window are hidden for the user. This means that only the structure of the event (the gate identifier and the event-offers) is displayed. If you have a constraint oriented specification this option is probably not useful, as all events will have the same structure and only differ in their predicates. But it is easier to see the structure of the simulation tree when the conditions are not shown.

Show extensions

Default: False.

Effect: The extensions of the variables and value identifiers are shown. This option is identical to the **Show extensions** option in the **Options** menu of the specification window, and is explained in more detail there.

Show intermediate results

Default: True.

Effect: During unfolding, the simulation tree is incrementally unparsed. That is, as soon as the unfolding of a node has been computed, this is reflected in the simulation tree window. If this option is switched off, this is not done. This saves considerable execution time (up to 50%) when both the simulation tree is large and the unfold depth is high.

4.2.6 **Misc** pull-down menu

Show statistics

Result: The SMILE Statistics window is popped up (see section 3.1).

Save EFSM as text

Result: A pop-up window appears in which a file name can be entered. When that is done, an extended finite state machine (EFSM) representation of the state machine is printed to the specified file. If the simulation tree has been completely unfolded (i.e. every

leaf node is either a deadlock node or is analyzed elsewhere), the transitions in the state machine which are not part of the simulation tree are not written to this file. If the simulation tree is not completely analyzed, the complete state machine is printed to a file. Any states which are not unfolded are printed as (generally very complex) behaviour expressions. The EFSM is printed as a plain, human-readable, LOTOS file. The original data types are *not* present in this representation. See **Save EFSM as .cr** for an option in which a complete representation is generated.

The generated EFSM is strong bisimulation equivalent with the simulated behaviour expression. It is structured as follows:

- the specification *name* is the name of the simulated process
- the behaviour expression in the definition block of the specification is a process instantiation of the first state of the state graph.
- all *expanded* states are denoted by a list of choice expressions, where each choice expression consists of an optional generalized choice operator, followed by an optional list of guards, followed by an action denotation or `exit`, followed by either `stop` or a process instantiation. example :

```
PROCESS state357 [ gate1 ] ( var: s ) : exit(s, bool) :=
  (choice x:nat [] [x isin var] -> g!x [x lt 4] ; state1[g])
  []
  (choice y:bool [] exit(var,y))
ENDPROC
```

- all states which are *not* expanded are denoted by a behaviour expression. If such states exist, the original process definitions are also added to the EFSM file (as a process call may occur in the unexpanded states).

Save EFSM as .cr

Result: This function is similar as **Save EFSM as text**, except that the EFSM is written in **.cr** format. This implies that the finite state machine can be handled by any other *lite* tool, if the specification name in the main *lite* window is set to the specified EFSM-filename. The generated EFSM file contains the finite state machine, the original processes (if the graph is not complete) and the original data type specification.

Save Tree

Result: A pop-up window appears in which a file name can be entered to which the simulation tree is dumped in ASCII format. The structure which is dumped to a file is identical to the structure displayed in the simulation tree window. This is especially useful for very large simulation trees, which can easier be checked on paper or with a text editor, than on a screen. Note that it might be even more efficient to use Tcl-commands to analyze the simulation tree.

4.2.7 Traversing the simulation tree

In addition to the commands in the pull-down menu, it is also possible to use the cursor keys to navigate through the simulation tree. The following functions are defined (only if the mouse-pointer is actually inside the simulation tree window):

First son

Key: This function is bound to the →-key (`R12`).

Result: The first son of the selected node is selected. If the selected node is not unfolded yet, this button will unfold it exactly once (the installed unfold-depth is not checked). This corresponds to the **One step** command in the **Execute** pull-down menu.

Last son

Key: This function is bound to the `End`-key (`R13`).

Result: The last son of the selected node is selected. If the selected node is not unfolded yet, this button will unfold it exactly once (the installed unfold-depth is not checked).

Father

Key: This function is bound to the ←-key (`R8`).

Result: The event leading to the father node of the selected node is selected.

Next sibling

Key: This function is bound to the ↓-key (`R14`).

Result: The next event, belonging to the expansion of the same simulation tree node, is selected. If the currently selected event is the last event of an expansion, the first event of the same expansion will be selected.

Previous sibling

Key: This function is bound to the ↑-key (`R8`).

Result: The previous event, belonging to the expansion of the same simulation tree node, is selected. If the currently selected event is the first event of an expansion, the last son of the previous sibling of the father node will be selected.

Root

Key: This function is bound to the `Home`-key (`R7`).

Result: The root node of the simulation tree will be selected.

4.3 SMILE Adt Interface window

This window consists of three parts: the specification part, the goal window, and the solutions window. The functions offered in each part will be discussed in separate sections.

4.3.1 ADT specification window

This window contains all kinds of information about the instantiated event, or about the ADT specification.

Source file

Parameters: None.

Result: The flattened data type definitions of the current specification are displayed in the 'source' window.

List Equations

Parameters: The current selection in the Source window. If nothing is selected there, then the current selection in the Goal window is used. If also nothing is selected there, then the previous selection is used.

Result : The parameter is supposed to be an operation identifier. All equations for that operation identifier are printed. Note that, due to overloading of operation identifiers, there may be more equations than expected.

List Operations

Parameters: See **List Equations**, above.

Result: All definitions for the selected operation definition are given. Due to overloading, there may be more than one.

Path

Parameters: The event which is currently being instantiated. See also the **Instantiate** command defined in the previous section.

Result: The path from the root node to the event is printed. This enables one to see where the variables in the goal are defined, and from which event they originate.

Remarks: This command is only enabled if the ADT-window is created (or refreshed) by the **Instantiate** command.

Free variables

Parameters: The resulting state of the instantiated event. If the Adt-Window is popped up using the **Adt Interface** command, the root node of the simulation tree is used as a parameter.

Result: All free variables of the parameter state, with the corresponding sortnames, are shown in the specification window. This can be useful to see which variables can be instantiated in the goal window. Note that only the variables of the resulting state are given, not all variables that have been used on the trace leading to the instantiated event!

Diagnostics : If no process is being simulated, an error message is given.

Find Warning

Result: The text in the ADT source window is scanned to see if any warnings with respect to the executability of the abstract data type specification are present. The text will automatically scroll towards the next warning. The warnings are explained in section 3.8.

Dismiss

Result: The Adt Interface window is popped-down.

4.3.2 Goal window

In the 'goal' subwindow predicates can be entered or edited. Predicates are separated by a ';' symbol. Overloaded value identifiers are not allowed. It must be possible to derive the sort of an identifier in the first predicate in which it is used, except if it is a value identifier introduced during simulation. In that case, it is assumed that that value-identifier has the same sort as in the simulation tree.

It is possible to indicate that a value identifier, or an operation, is of a specific sort using the LOTOS `ofsort` construct, which may be abbreviated to ':'. The edit-commands which can be used are the standard Motif text-edit commands. Notably, the middle mouse button can be used to insert text from the clipboard, and typing new text removes the selected text.

Solve goal

Parameters : The typed in goal.

Result The goal is first checked on syntax and static semantics errors. If that succeeds, the narrowing tool is called on the goal and the result is placed in the Solution window. The result may be one of the following:

- A solution. This solution is then printed in the Solution window. The 'Next Solution' and 'Use' buttons are enabled.
- `[Maybe no solution exists]`. The narrowing tool was not able to find solutions because the specification contains some semi-constructors (functions used as constructors). See also section 3.8.
- `[No solution exists]`. There exists no solution for the conjunction of the predicates.
- `[Found all solutions]`. The goal is a tautology (i.e. is always true), or all solutions are found (after having pressed the **Next Solution** button).
- `[Exception: Unknown solution]`. The tool is not able to find a solution for the predicates. A more detailed description of the reason for this is given in the status line. Reasons are: Argument stack overflow, dump overflow, heap overflow, recursion overflow, trail stack overflow, variable symbol table overflow, maximum number of narrowing steps reached, and maximum number of instantiations reached. The last two error messages occur most, and have to do with the structure of the specification. If you frequently get one of the other messages, which might happen on *very* large specifications, it could help to increase some of the built–in SMILE tables. How to do that is described in the next section.

Diagnostics: If the static semantics check fails on the given goal, an error message is printed in the status line and the errors are indicated using comments in the affected equation(s).

Rewrite goal

Parameters The typed goal.

Result The goal is checked with respect to the static semantics. Then the complete goal is rewritten into normal form.

Diagnostics An error message is printed if the goal is not correct with respect to the static semantics.

Add to Tree

Parameters The goal and the event which is instantiated in the SMILE main window. If the Adt Interface was not called using the **Instantiate** button, the predicates are added to the root node of the simulation tree.

Result The goal is checked with respect to the static semantics. If that does not fail, the event is copied in the simulation tree (with its resulting behaviour), and the new predicates are added to that particular event. Then the result node of the state is, recursively, re-examined to check whether all the events are still possible. Impossible events are pruned from the tree.

Diagnostics If there is no process under simulation, an error message is given. An error message is also printed if the goal is not correct with respect to the static semantics.

Substitute in Tree

Parameters The goal and the event which is instantiated in the SMILE main window. If the Adt Interface was not called using the **Instantiate** button, the predicates are added to the root node of the simulation tree.

Result The goal is checked with respect to the static semantics. If that does not fail, the event is copied in the simulation tree (with its resulting behaviour). The goal is viewed (as much as possible) as a list of substitutions for specific variables. These substitutions will be performed on the copied event. The result node of the event will be re-examined to check whether all the events are still possible. Impossible events are pruned from the tree.

Diagnostics If there is no process under simulation, an error message is given. An error message is also printed if the goal is not correct with respect to the static semantics.

4.3.3 Solution window

Next solution

Parameters: None.

Result: The next solution for the goal is given. The output is described above, under the command **Solve goal**.

Remarks: This function is only enabled if there was a previous solution and if the unfolding is not busy.

Use

Parameters: The selected solution.

Result : the solution is added to the goal. This is useful if the narrowing tool loops in producing solutions for only one variable. Then a solution for this particular variable can be added to the goal and the goal can be solved again. If any other solutions exist, they will be found in that case.

Diagnostics Tautologies, i.e. solutions of the form `v = v` cannot be added to the goal.

4.4 Info Window

This window can be manipulated using normal editing commands. Specific commands are:

Dump to File

Parameters : The filename typed in the area right next to the command button.

Result : The contents of the info window are written to the specified file.

Dismiss

This commands pops down the info window.

4.5 Command Window

This window contains only one command:

Dismiss

This commands pops down the command window.

The edit-area can be used in a `xterm`-like fashion: only the last line is editable. The prompt `Smile>` changes into > if the tool thinks that the command you entered is incomplete. The commands that can be given in the text area are outlined in more detail in chapter 5.

Chapter 5

Programming SMILE

In the command window any Tcl command can be typed. A number of commands have been added to Tcl that give access to the simulation tree and the basic functions of SMILE. Note that these functions are still quite experimental. It is easily possible to extend the (basic) command set with additional commands using the Tcl procedure definitions (see for instance section 3.2 or section 3.7 for some examples).

The interface between the simulation tree and the command-line window is through the use of integer values. Each event is numbered. The commands return a Tcl-list of these numbers, that correspond to the events produced by the command. A 'state' is indicated by the event that has that particular state as end-point. The initial state is indicated with the special event identifier '0'.

The interface between SMILE and Tcl consists of a number of variables and commands. We first list all special variables, and subsequently all commands that are added to Tcl.

5.1 Variables with special meaning

Integer variables:

current_event

This is the link between the command-line interface and the specification window. Reading this variable returns an identifier for the selected event (an integer); setting it sets the selection in the specification window.

unfold_depth

the depth of the tree generated by a single unfold-command

narrowing_power

an estimate of the amount of time that is spent on the fly by the adt-evaluator. Increasing it implies that it might find solutions where it could not find them earlier, but it also implies that it takes longer before a nonterminating evaluation stops.

unguarded_recursion

the number of times a recursive call that is not guarded by at least one action prefix expression is evaluated.

recursion_cost

The cost of performing a recursive process instantiation. Used during goal-oriented evaluation, for the determination of the cost of reaching the goal from the current state.

reanalyze_cost

The cost of re-evaluating a state that has already been analyzed, but with different values of the free variables. Also only interesting for goal oriented evaluation.

The effect of the last two variables is explained in more detail in section 3.7.

Boolean variables:

unparse_incremental

Show intermediate results during the generation of (a large) simulation tree.

normalize_states

Normalize the state expressions after unfolding.

partially_evaluate

When performing goal-oriented execution, try to generate only the interesting events.

find_unique_solutions

evaluate all predicates twice. If only one solution exist, use this solution instead of the predicate.

create_efsm

Build an extended finite state machine as a side-effect of the creation of a simulation tree.

breadth_first

Search strategy in the simulation tree. Especially useful for target-driven evaluation, and non-terminating evaluations (yes, the tool stops when it almost runs out of memory).

postpone_reevaluation

influences the strategy used by goal-oriented evaluation. If 'partially_evaluate' is set, the tool might not compute the events that it should have (this might seem strange, but never mind). It detects this, and queues these states for re-evaluation. If this option is set, re-evaluation is only performed if no 'new' states are left. If it is not set, the system uses its built-in strategy no matter whether it is a re-evaluation or not. This built-in strategies is based on a static computation on the distance towards the target behaviour expression (or action denotation). See also section 3.7.

Most of these variables are connected to the input lines or toggle options of the X-Window interface. I.e. setting an option is immediately reflected in the corresponding variable (and the other way around).

5.2 Commands

The following commands are added to the standard Tcl-command set:

simulate

Parameters : a process identifier

Return type :

Description : Starts the simulation of the given process. Starting the simulation has the following side-effects:

- all parsed action denotations are removed
- all existing simulation resuls are removed

It is equivalent to the **Start** command in the **Simulate** menu.

ignore

Parameters : A Tcl list of process identifiers

Return type :

Description : if no parameters are given, the list of all ignored process identifiers is shown. otherwise, each process in the process identifier list will not be used during unfolding. See also the **Ignore** command in the previous chapter.

use

Parameters : A Tcl-list of process identifiers

Return type : re-use the list of parameter processes during unfolding. This undo's the effect of the `ignore` command.

unfold

Parameters : An event-identifier.

Return type : A Tcl-list of event identifiers.

Description : unfold the given node. Similar as the Unfold command of the simulation tree window. It uses the same option-setings, and returns the list of children of ONLY the parameter node. This might be considered as a bug (because potentially a tree of depth `$unfold_depth` has been generated).

expand

Parameters : an event identifier.

Return type : a list of event identifiers.

Description : unfolds the given state, but only one step deep. Returns the children of that state.

children

Parameters : an event identifier.

Return type : a list of event identifiers.

Description : returns the list of children if the state has been unfolded, if it has not been unfolded, it returns an error.

father

Parameters : an event-identifier

Return type : an event-identifier

Description : returns the event identifier corresponding to the source node of the parameter event. It returns an error if the parameter is 0.

show_event

Parameters : event-identifier

Return type :

Description : highlights the event in the X11 user interface. Similar as setting the variable `current_event`.

identical_node

Parameters : an event-identifier.

Return type : event-identifier.

Description : if the given state is equivalent with another state, the corresponding state is returned. (See also the flags, below.) It is equivalent to the function **Identical node** in the X11 interface.

overlap

Parameters : 2 event identifiers

Return type : boolean

Description : returns true if both events denote an overlapping set of transitions, according to the LOTOS synchronization rules (i.e. `g?x[x>4]` overlaps with `g?y[y<10]`)

match

Parameters : 2 event identifiers

Return type : boolean

Description : returns True if both events denote the same set of transitions. This is implemented using structural comparison modulo variable names. Therefore, `g ?y` matches `g?z`, but `g?x[x<10]` does not match with `g?x[x<10, x<20]`.

parse_action

Parameters : string

Return type : action-identifier.

Description : It parses 'string' into an action. This is covered in more detail in section 3.7.

show_action

Parameters : action-identifier

Return type : a Tcl list.

Description : reverse operation of `parse_action`. It returns a Tcl-list of three elements: the first is the gate identifier, the second is a Tcl-list containing the experiments, and the third component is a Tcl-list containing the predicate. An experiment is a triple, where the first element is either a '?'-symbol or a '!'-symbol, the second element is a value expression, and the third element is the sort of the value expression. A predicate is a singular list, containing only a string that corresponds to the unparsing of the predicate.

goto_target

Parameters : a target description (see section 3.7), an event-identifier, and an optional integer reflecting the maximum length of the generated trace.

Return type : a Tcl-list of event identifiers.

Description : this command evaluates the second parameter, until the target specified as first parameter is reached. The optional third parameter denotes the maximal number of nodes that may be evaluated by this command. If it is not present, there is no limit. A target is described in more detail in section 3.7. The tool first generates an initialization sequence for the first target-element starting at the given state, and then tries to generate an initialization sequence for the next element of the target, starting at the last state of the first initialization sequence, etc. The tool uses a depth-first search between each target-element. The search strategy for each subtarget can be controlled

by the breadth_first option (see above). For instance, a sequence of three input events starting from the initial node of the system can be computed as follows:

```
set a [parse\_action "input *"]
goto\_target {$a $a $a} 0
```

`goto_target` returns the last event of the computed initialization sequence. In the rare case that the last evaluation results in multiple events that reach the desired target, all these events are returned.

next

Parameters :

Return type : a list of event identifiers

Description : performs an explicit backtrack step in the goal-oriented execution mechanism to produce the next possible initialization sequence for the given target. See also section 3.7.

5.3 Predicates

The following commands all accept an event/state as parameter, and return a flag (a boolean value) that might have been set for that event/state. These predicates can be used to analyze the events produced by the `unfold` or `expand` command, and in such a way allow one to program one's own evaluation strategy.

analyzed_elsewhere

True if the state is identical to another state, but the other state is not on the path from the initial state to the parameter state

recursion_detected

True if the state is identical to some other state that lies on the path from the initial state to the parameter state

sync_failed

True if partial evaluation of this state may not have computed all events.

hidden_by_user

True if this particular event has been hidden by the user (hide-command).

goal_active

True if this event is the last event in an initialization sequence that has been computed by the 'goto-target' command.

predicates_solved

True if the adt-evaluator has proven that there exists a substitution for the free variables in this event. If it is False, evaluation has not been completed. Increasing the narrowing power might help.

partially_expanded

True if only a part of all children have been generated. It implies 'unfolded'

unfolded

True if the given state has been unfolded.

5.4 Some example Tcl-programs

To be provided

Chapter 6

Customizing SMILE

6.1 Customizing the display

It is easy to change the appearance of (a part of) SMILE. This can be done by setting some resources in the X window resource database using xrdb [1]. All resources are described in the file `$LITEDIR/$LITEARCH/lib/Smile`. The most useful resources are:

resource name	class	default value
smile*fontList	Font	-*-helvetica-bold-r-*-*-12-*-*-*-*-*-*-*
smile*background	Pixel	snow2
smile*foreground	Pixel	black
smile*XmPushButton*background	Pixel	snow1
smile*XmToggleButton*background	Pixel	snow1
smile*XmTextField*background	Pixel	snow1
smile*XmCascadeButton*background	Pixel	snow1
smile*XmText*background	Pixel	white
smile*XmtCli*background	Pixel	white

The Motif widget classes `XmPushButton`, `XmToggleButton`, `XmTextField`, `XmLabel`, `XmText` and `XmCascadeButton` are used by SMILE, as well as the Motif Power Tools widget class `XmtCli` (the command-line interface is built using that widget). These classes can therefore be used in a user's resource file specification. The class names can be used to set different resources for different classes. If you want a specific button or text area to have another default appearance, please check the general `Smile` file for the actual name of that particular widget. The Adt Interface window may be addressed by setting a resource for `smile.adt_object.`, the main window by `smile.smile_object.` So for instance

```
smile.smile_object*XmCascadeButton*foreground: ForestGreen
smile.adt_object*XmPushButton*foreground: Red
```

sets the foreground colours of the command buttons in the Adt Interface to red, and of the pull-down menu-buttons in the main SMILE window to green.

6.2 Memory management

Efficient memory management has been a key issue in the design of SMILE. Care has been taken to use some good default values to make execution of medium sized LOTOS specifications ($\approx$ 3,000 lines) as efficient as possible. For that reason, a number of hash tables are used internally. However, for much larger specifications (or very complex specifications) it might be necessary to increase the sizes of the hashtables. (Also, if you use small specifications on a small machine, it might be useful to decrease the sizes). SMILEuses 4 different hashtables. For this reason, SMILE reads a `.smilerc` file. This file can either be placed in the user's home directory, or in the working directory of the user (where the latter takes precedence over the first). In this file, the default sizes of the hashtables can be overruled by specifying one of the following values:

Resource	default value	affected
SpecificationTable	50	symbol tables, initial specification
ExpansionTable	150	EFSM, Simulation tree
TempTable	20	temporary, cleared every expansion step
NarrowingTable	5	temporary, cleared every call to narrowing module

For instance, the size of the SpecificationTable can be set to 200 by placing the following line in the `.smilerc` file in your home directory:

```
SpecificationTable      200
```

The actual sizes of the hashtables are a thousand times bigger than the actual values. Note that increasing the sizes of hashtables when it is not necessary, increases the sizes of some tables, and therefore the memory usage of the tool. This may lead to swapping of memory pages, and therefore to a decreasing performance of the tool. On the other hand, if hashtables are quite overloaded (a lot of buckets with more than, say, 8 elements), it can be quite rewarding in terms of execution speed to increase the size of the affected hashtable.

Some rudimentary information about the number of free entries in the different hashtables can be generated by enabling the **Memory** command. This can be done using resources. The following resource (see the Unix manual page of **xrdb**) can be set:

```
smile*mem_info.wcManaged: True
```

This will give you an additional command in the `Misc`-pull down menu of the specification window. Pressing this button will give you an enormous amount of figures about the memory consumption of SMILE. This contains also information about the relative sizes and the amount of free entries in the hashtables.

6.3 Narrowing constants

It is possible to set a number of constants that have to do with the narrowing interpreter. This can also be done in the `.smilerc` file(s). The following constants can be set:

Resource	default value	affects
TrailStackSize	30000	size of trailstack
HeapSize	400000	size of the heap
CodeSize	600000	amount of space for compilation 30 per equation needed.
MaxRecursion	1000	max. number of recursive steps. Used to handle goals like `succ(x) = x`
MaxOpsyms	10000	max. number of operators
MaxVarsyms	2000	max. number of variables in a goal
MaxInstantiations	10000	max. number of substitutions for **Solve Goal**
MaxNarrSteps	1000000	max. number of narrowing steps for **Solve Goal**
SortNarrSteps	500	Number of narrowing steps used when sorting individual equations in a goal. Used to determine the evaluation order.
SortInstantiations	30	Instantiations used during goal sorting
SortSolutions	15	Solutions used during goal sorting
TestNarrSteps	10000	Amount of steps used during report generation
TestInstantiations	100	Reduced limit for report generation
MaxSolutions	20	limit for termination test (report generation)

Note that the value of the constants is set during the startup of SMILE. The **Size**–constants increase some internal tables, and therefore affect the amount of memory used by the tool.

6.4 Setting constants for everybody

It is also possible to set some of the hash-table sizes or narrowing constants permanently, for everybody that uses the tool. That can be done by placing the constant definitions in the file `$LITEDIR/$LITEARCH/lib/smile/.smilerc`, or in the file `$SMILELIB/.smilerc` (the latter takes precedence over the first). These definitions are overruled by the values in the user's own `.smilerc` file(s), but overrule the built-in default values.

6.5 Out of Memory

When SMILE runs out of memory, it will print an error message in the error line, and will try to gently stop the current execution. In normal cases, this will succeed, and the user is able to either free some system memory (by killing other applications), or to analyze the results of the current session in the tool. This is done using a reserved block of memory. Sometimes, however, this block of memory is not enough to finish the current operation. In that case, SMILE will exit (unfortunately...).

6.6 Adding commands to SMILE

It is possible to program your own commands in Tcl, using any of the SMILE-specific additions to Tcl that are described in the previous section. To be able to use your own collection of pre-defined Tcl-scripts, SMILE reads the following files *after it has read the specification*, but before control is given to the X11-interface:

1. `$LITEDIR/$LITEARCH/lib/smile/smile-init.tcl`
2. `$SMILELIB/smile-init.tcl`
3. `$HOME/smile-init.tcl`
4. `smile-init.tcl`

Any procedures can therefore be automatically defined on start-up. It is even possible to automatically start the evaluation of the specification using a script (because the files are executed after the specification has been read).

It is also possible to add a new button to the X11-interface that calls a Tcl-function. This is, however, slightly more complicated, because the file `$LITEDIR/$LITEARCH/lib/Smile` must be edited for that purpose. Basically, it is necessary to add a new `XmPushButton`-widget to the hierarchy (somewhere), and use the callback function `TclCommandCB` where you have to supply the name of the command that has to be executed by the Tcl-interpreter as a parameter of the callback. If you do not understand this, it is probably a good idea to ask your local X-guru to help you out. The `Smile` resource file, however, could already help you a lot in understanding how the procedure of creating a graphical resource file using `Wcl` works.

The result-value of the new command is written to the command-line interface of SMILE.

Acknowledgements

First versions of SMILE were built in the ESPRIT II project LOTOSPHERE. We would like to thank all people in workpackage 2 of that project who contributed to the design of the tool. In particular, Peter van Eijk(UT) and Jean-Bernard Saint(INRIA).

The narrowing tool has been designed and implemented by Dietmar Wolz from the Technical University of Berlin. Without his contribution, SMILE would probably not exist at all. All other parts of SMILE have been designed and implemented by Henk Eertink (University of Twente).

SMILE is completely implemented (except for the user interface and the narrowing interpreter) using the **termprocessor**, a metatool designed and implemented by Axel Belinfante and Peter van Eijk. This eased the implementation a lot.

The user interface of SMILE is designed using the Widget Creation Library, version 2.5. Wcl is a public domain library that eases the implementation of X11 based user interfaces. It has been developed by David E. Smyth, Jet Propulsion Labs, California Institute of Technology, Pasadena, USA; and is available as contributed software in the X11R5 distribution from MIT. SMILE also uses version 1.1.5 of the Motif widget set, as well as a few widgets and convenience routines supplied in the Motif Power Tools package, available as shareware from Dovetail Systems, Inc. The command line interface is based on the Tool command language, designed by John Ousterhout at Berkeley, available also as public domain software.

SMILE is copyrighted software from the University of Twente, TIOS group, the Netherlands.

More Information

For more information about the functionalities of SMILE, do not hesitate to contact

```
LOTOS Tool development group
Dept. of Computer science, TIOS
University of Twente
PO Box 217
7500 AE Enschede, The Netherlands
Lotos.Tools@cs.utwente.nl
```

Bibliography

[BE 93] Ed Brinksma, Henk Eertink, *Goal driven LOTOS execution*, *in* Protocol Specification, Testing, and Verification 13, Liege (Belgium), North-Holland, 1993.

[Eer 94] Henk Eertink, *Simulation Techniques for the Validation of LOTOS Specifications*, PhD thesis, University of Twente, 1994 (to appear).

[EW 92] H. Eertink, D. Wolz, *Symbolic Execution of LOTOS Specifications*, proceedings of the FORTE '92 conference, Lannion, nov. 1992, North-Holland, 1992.

[Wolz 91] D. Wolz, *Design of a Compiler for Lazy Pattern Driven Narrowing*. Proceedings of the 7th international workshop on the specification of abstract data types, Springer LNCS 534, 1991.

[Ous 90] J.K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, pages 133–146, 1990.

[Ous 94] John K. Ousterhout. *Tcl and Tk*. Addison-Wesley, 1994 (to appear). Also available using anonymous FTP. See newsgroup `comp.lang.tcl`.

Index

–, 36
.., 23
Unguarded Recursion, 40

Add to Tree, 49
Adt Interface, 34
adt warnings, 29
analyzed elsewhere, 5, 36
analyzed_elsewhere, 57

Breadth-first, 40
breadth-first, 28
breadth_first, 53
Build EFSM as well, 39

children, 55
Command Interface, 34
Complete EFSM, 38
cost function, 27
Create EFSM, 16
create_efsm, 53
current_event, 20, 51
customization, 59

depth-first, 28
Disable state normalization, 39
Dismiss, 47, 50
display, 3
Display menu, 43
Dump to File, 50

EFSM, 14
error line, 3
event identifier, 19
Event menu, 40
executability, 29
expand, 54

Father, 45
father, 19, 55
Filter Inference system, 39
Filter inference system, 24, 27
Find action prefix, 41
Find Warning, 47
find_unique_solutions, 52
finite state machine, 14
First son, 45
Free variables, 47

goal, 8
    editing, 47
goal_active, 58
Goto target, 37
goto target, 25
goto_target, 25, 56
guardedly well-defined, 36

Help, 35
hidden_by_user, 57
Hide, 40
Hide rest, 41

Identical node, 42
identical_node, 55
Ignore, 23
ignore, 54
    menu, 33
    reset, 33
    set, 33
    show, 33
Ignore menu, 33
incremental unparsing, 43
Instantiate, 22, 42
instantiated event, 5

Last son, 45
List Equations, 46
List Operations, 46

match, 56
Misc menu, 43

narrowing, 28
narrowing power, 35
narrowing_power, 51
next, 26, 57
Next sibling, 45
Next solution, 49
Next Target, 26, 37
normalize_states, 52

One step, 37
Options menu, 33
overlap, 55

parse_action, 25, 56
partial, 23
    expansion, 24
partially_evaluate, 27, 52
partially_expanded, 58
Path, 47
postpone_reevaluation, 28, 53
predicates_solved, 58
Previous sibling, 45

Quit, 32

reanalyze_cost, 28, 52
recursion detected, 5, 36
recursion_cost, 27, 52
recursion_detected, 57
Reset, 32, 33
Rewrite Always, 39
Rewrite event, 41
Rewrite goal, 48
Rewrite Minimal, 39
Root, 46
Run Test, 21, 32

Save EFSM as .cr, 44
Save EFSM as text, 43
Save Tree, 44
Set, 32, 33
Show, 32, 33
Show behaviour, 33
Show extensions, 33, 43
Show intermediate results, 43
Show menu, 42
Show node, 42
Show path, 42
Show predicates, 43
Show statistics, 43
show_action, 56
show_event, 20, 55
Simulate, 31
simulate, 19, 53
simulation tree, 14
Solve goal, 48
Source File, 29
Source file, 46
specification window, 3
state identification, 5, 16
State menu, 42
statistics, 16
Stop, 38
Substitute in Tree, 49
Substitute Unique solutions, 39
sync_failed, 57

target
    action, 24
    behaviour, 23
    menu, 32
    next, 26
    reset, 32
    set, 32
    show, 32
Target menu, 32
Trace behaviour expressions, 40

Unfold, 35
unfold, 19, 54
    depth, 35
unfold_depth, 51
unfolded, 19, 58
unguarded_recursion, 52
unparse_incremental, 52
Until stable, 38
Use, 49
use, 54